chapter-02-01

March 14, 2025

Run in Google Colab

1 Clustering Techniques

1.1 Introduction

The Problem

- In the context of unsupervised learning, the clustering problem involves **grouping a set of data points into subsets**, or "***clusters***", where points within the same cluster are more **similar** to each other than to those in other clusters.
- The goal is to identify inherent patterns or structures within the data **without any prior knowledge of group labels**.
- Clustering is widely used in various applications like market segmentation, social network analysis, organizing large data libraries, and anomaly detection in diverse fields ranging from bioinformatics to finance.
- This task distinguishes itself from other machine learning paradigms by its lack of reliance on pre-assigned labels or categories. The algorithms must discern patterns and categorizations solely based on the inherent attributes present within the data.

Key aspects

1. **Objective**: The primary objective is to partition the data into clusters such that the points within each cluster are more similar to each other than to points in other clusters. However, the definition of ***"similarity"*** can vary depending on the specific clustering algorithm and the nature of the data.
2. **Unlabeled Data**: Unlike supervised learning, where the algorithm is provided with labeled examples to learn from, in unsupervised learning, the data is unlabeled. This means that the algorithm must discover patterns or structures solely based on the inherent properties of the data itself.
3. **Algorithm Selection**: There are various clustering algorithms available, each with its own approach to defining and identifying clusters. Common algorithms include K-means, hierarchical clustering, DBSCAN (Density-Based Spatial Clustering of Applications with Noise), Gaussian Mixture Models (GMM), and more. The choice of algorithm depends on factors such as the nature of the data, the desired number of clusters, and computational considerations.
4. **Evaluation**: Evaluating the quality of clustering results can be challenging in unsupervised learning since there are no ground truth labels to compare against. However, there are several

metrics and techniques, such as silhouette score, Davies-Bouldin index, and visual inspection, that can help assess the effectiveness of clustering algorithms.

Similarity

- A fundamental aspect of the clustering problem is the establishment of a criterion for **similarity**.
- The efficacy of clustering largely hinges on the appropriateness of the chosen similarity or distance metric (such as Euclidean or Manhattan distances). -
- These metrics are pivotal in assessing the likeness between data points, thereby guiding their aggregation into cohesive groups.

Cluster Morphology

- An intriguing facet of clustering is the diversity in the morphology of clusters; they can vary widely in shape, size, and distribution, presenting a significant analytical challenge, especially in datasets with complex structures or overlapping cluster boundaries.
- Another critical consideration is the determination of the optimal number of clusters, a parameter that significantly influences the outcome of the clustering process; some algorithms demand a pre-specified number of clusters, while others are capable of autonomously ascertaining this number.

Applications

The clustering problem manifests in various practical applications, each highlighting its versatility and significance. In the commercial sector, clustering is instrumental in customer segmentation, enabling businesses to categorize customers based on purchasing patterns, demographics, or preferences, thereby enhancing marketing strategies and product development. In the domain of image processing, clustering facilitates image segmentation, a process where pixels with similar features are grouped, aiding in tasks such as object recognition and medical imaging.

Applications in Finance

Clustering techniques in finance are utilized to uncover hidden patterns, group similar entities, and inform decision-making processes. Their applications span various domains within the financial sector, providing valuable insights and optimization strategies. Here's a brief overview of some of the main applications:

1. **Portfolio Management:** Clustering is used to categorize stocks, bonds, and other assets into groups based on their returns, volatility, or other financial metrics. This helps in diversifying portfolios, understanding asset behaviors, and identifying correlation patterns among investments, leading to more informed asset allocation and risk management strategies.
2. **Fraud Detection:** By clustering accounts or transactions based on similarities in behavior or attributes, financial institutions can identify unusual patterns or anomalies that deviate from the norm. This is critical in detecting potential fraud, money laundering, or other illegal activities, enhancing the security and integrity of financial systems.
3. **Customer Segmentation:** Financial entities employ clustering to segment customers based on various criteria such as spending habits, income levels, investment preferences, or risk tolerance. This enables personalized marketing, tailored financial advice, and customized product offerings, improving customer satisfaction and loyalty.

4. **Credit Scoring:** Clustering helps in grouping borrowers with similar credit characteristics or behaviors, aiding in the development of more accurate credit scoring models. This can improve the assessment of credit risk, leading to more informed lending decisions and the potential for reduced default rates.
5. **Market Structure Analysis:** Clustering techniques are used to analyze and understand market structures by grouping similar financial instruments, identifying market segments, or understanding the behavior of market participants. This can inform trading strategies, market regulation, and the identification of market trends or cycles.
6. **Risk Management:** By clustering financial instruments or entities based on risk characteristics, institutions can better understand and manage systemic risks, credit risks, and operational risks. This aids in the development of strategies to mitigate exposure and protect against market volatility or economic downturns.
7. **Operational Efficiency:** Clustering can also be applied to optimize operations within financial institutions by identifying similar processes, customer service inquiries, or transaction types. This can lead to more efficient process management, resource allocation, and customer service strategies.

Overall, clustering techniques in finance provide powerful tools for data analysis, decision-making, and strategy development, helping to navigate the complexities of the financial markets and improve organizational performance.

Challenges and Difficulties

- Although the clustering problem has numerous applications, it is not without its difficulties.
- First, **the definition of a cluster is not fixed universally but varies significantly depending on the context**. Different algorithms identify clusters in different ways. For instance, some algorithms aim to find instances grouped around a central point, termed a centroid, while others look for densely populated areas, resulting in clusters of various shapes.
- Moreover, **selecting the most suitable clustering algorithm is a complex task**, heavily influenced by the characteristics of the data and the specific needs of the clustering problem.
- Additionally, clustering high-dimensional data poses particular challenges due to the '***curse of dimensionality***', where the abundance of features causes the data to be sparse, making traditional distance measures less reliable.

Clustering is Subjective

- Scalability is another crucial factor, especially when dealing with large datasets, necessitating algorithms that can maintain performance without a significant compromise in accuracy.
- Furthermore, the interpretation of clustering results often lacks straightforwardness and typically requires domain-specific expertise for validation and comprehension.

1.2 Four different types of clustering methods

1. The first one is **centroid-based** clustering. Each cluster is represented by a centroid which derives clusters based on the "distance" of the data point to the centroid of the clusters. One of the most widely used centroid-based algorithms is ***k-means***.

2. The second one is **connectivity-based** clustering. The clusters are defined by grouping the nearest neighbor based on distance between the data points. The idea is that nearby data points are more related than other points farther away. Two examples of this class of methods are ***Hierarchical*** Clustering and ***Spectral*** Clustering.
3. The third one is density-based clustering. Clusters here are defined by areas of concentrated density. In this Lesson we discuss the ***DBSCAN*** Clustering Method which belong to this class.
4. The fourth one is **distribution-based** clustering. In this method, each cluster belongs to a normal distribution. The idea is that data points are divided based on probability of belonging to the same normal distribution. The ***Gaussian Mixture Model*** is one of the most important example of this kind of clustering methods.

1.3 K-Means

1.3.1 Definitions

- *k*-means clustering is one of the simplest and popular unsupervised machine learning algorithms. The objective of K-means is simple: group similar data points together and discover underlying patterns.
- To achieve this objective, K-means looks for a fixed number (k) of clusters in a dataset.
- You'll define a target number ***k***, which refers to the number of centroids you need in the dataset. A centroid is the imaginary or real location representing the center of the cluster.
- Every data point is allocated to each of the clusters through reducing the in-cluster sum of squares.

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the **nearest** cluster, while keeping the centroids as small as possible. The 'means' in the K-means refers to averaging of the data; that is, finding the centroid.

How the K-means algorithm works

To process the learning data, the K-means algorithm starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids. It halts creating and optimizing clusters when either:

- The centroids have stabilized — there is no change in their values because the clustering has been successful.
- The defined number of iterations has been achieved.

Source: Wikipedia

A Distance Measure

For clustering we need a distance measure. The simplest distance measure is the Euclidean Distance measure:

$$Distance = \sqrt{(x_B - x_B)^2 + (y_B - y_A)^2}$$

Multidimensional Distance

In general when there are m features the distance between P and Q is

$$d = \sqrt{\sum_{j=1}^{m} \left(\nu_{pj} - \nu_{qj}\right)^2} \tag{1}$$

where ν_{pj} and ν_{qj} are the values of the $j-th$ feature for P and Q

Local Vs Global Optima

Unfortunately, although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): this depends on the centroid initialization. A possibile solution is to run the algorithm multiple times with different random initializations and keep the best solution. This is controlled by the n_init hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier actually runs 10 times when you call `fit()`, and `Scikit-Learn` keeps the best solution.

1.3.2 Example 1 - Synthetic Dataset

- Let's see the steps on how the K-means machine learning algorithm works using the Python programming language.
- We'll use the Scikit-learn library and some random data to illustrate a K-means clustering simple explanation.
- Let's start to code ...

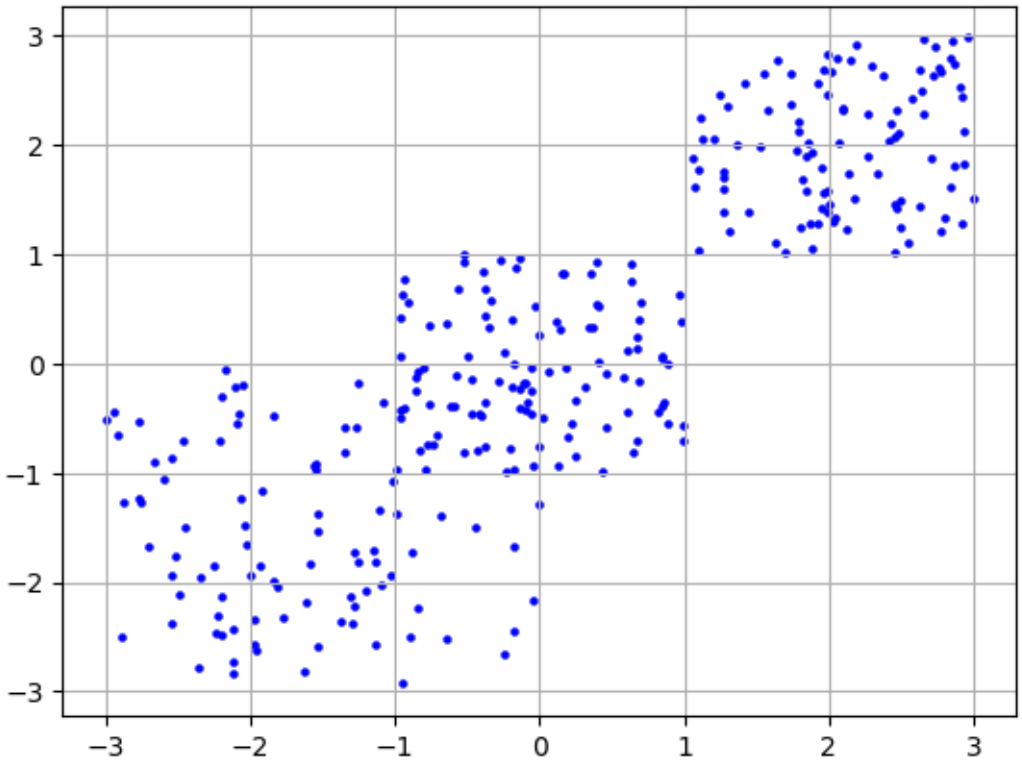
[1]:
```
# Import pandas library for data manipulation and analysis.
import pandas as pd
# Import numpy library for numerical operations on arrays and matrices.
import numpy as np
# Import matplotlib's pyplot module for creating static, interactive, and␣
 ↪animated visualizations in Python.
import matplotlib.pyplot as plt
# Configure Jupyter notebook to display matplotlib plots inline (directly in␣
 ↪the notebook cells).
%matplotlib inline

# Import the KMeans clustering algorithm from scikit-learn's cluster module.
from sklearn.cluster import KMeans


# Import the warnings library.
import warnings
# Configure warnings to ignore them, preventing them from being displayed. This␣
 ↪is often used to suppress deprecation,
# runtime, and syntax warnings that do not affect the execution of the program.
warnings.simplefilter('ignore')
```

Here is the code for generating some random data in a two-dimensional space:

[3]:
```
# Generating random numbers in the range [-3, 0) for 300 rows and 2 columns
rand1 = -3 * np.random.rand(300, 2)
# Generating random numbers in the range [1, 3) for 100 rows and 2 columns
rand2 = +1 + 2 * np.random.rand(100, 2)
# Generating random numbers in the range [-1, 1) for 100 rows and 2 columns
rand3 = -1 + 2 * np.random.rand(100, 2)

# Assigning rows 100-199 of rand1 with values from rand2
rand1[100:200, :] = rand2
# Assigning rows 200-299 of rand1 with values from rand3
rand1[200:300, :] = rand3
# Assigning the concatenated matrix to variable X
X = rand1
```

The code in the next cell, generates a scatter plot using the data stored in the variable `X`. Each row of `X` represents a data point with two features. The code selects the first column of `X` as x-coordinates and the second column as y-coordinates for plotting. It sets the size of markers to 5 and their color to blue. Finally, it adds grid lines to the plot and displays it.

[4]:
```
# Scatter plot of the data points in X
# X[:, 0] selects the first column of X (x-coordinates)
# X[:, 1] selects the second column of X (y-coordinates)
# s = 5 sets the size of the markers to 5
# c = 'b' sets the color of the markers to blue
plt.scatter(X[:, 0], X[:, 1], s=5, c='b')

# Displaying grid lines on the plot
plt.grid()

# Showing the plot
plt.show()
```



This give us three sets. Now we are going to perform K-means clustering using the `KMeans` algorithm from scikit-learn. It creates a `KMeans` object with `n_clusters=3`, indicating that it should cluster the data into three groups. The parameter `n_init='auto'` specifies that the algorithm should automatically determine the number of initializations to perform. Finally, it fits the KMeans model to the data stored in `X`, assigning each data point to one of the three clusters.

[5]:
```
# Creating a KMeans object with 3 clusters
# n_clusters=3 specifies the number of clusters to create
# n_init='auto' specifies the number of initializations to perform␣
 ↪(automatically determined)
Kmean = KMeans(n_clusters=3, n_init='auto')

# Fitting the KMeans model to the data stored in X. The fit method of the␣
 ↪KMeans object performs
# the training process, where the algorithm learns the optimal cluster␣
 ↪centroids based on the input data.
Kmean.fit(X)
```

[5]:
```
KMeans(n_clusters=3, n_init='auto')
```

In this case, we arbitrarily gave `k` (`n_clusters`) an arbitrary value of 3.

After the training we can start to check the results. First of all retrieve the cluster centers (centroids)

calculated by the KMeans clustering algorithm. These cluster centers can be used for various purposes, such as visualizing the clusters, analyzing the characteristics of each cluster, or making predictions for new data points by assigning them to the nearest cluster center.

[6]:
```
# The variable centers is assigned the result of accessing the cluster_centers_␣
 ↪attribute
# of the Kmean object. This attribute stores the coordinates of the cluster␣
 ↪centers after
# the KMeans algorithm has been fitted to the data using the fit method.
centers = Kmean.cluster_centers_
for k in range(len(centers)):
    c = ["{0:0.4f}".format(x) for x in centers[k]]
    print("Coordinate for center ", k, " : ", c)
```

```
Coordinate for center  0  :  ['2.0965', '1.9908']
Coordinate for center  1  :  ['-1.7623', '-1.6536']
Coordinate for center  2  :  ['-0.0677', '-0.0760']
```

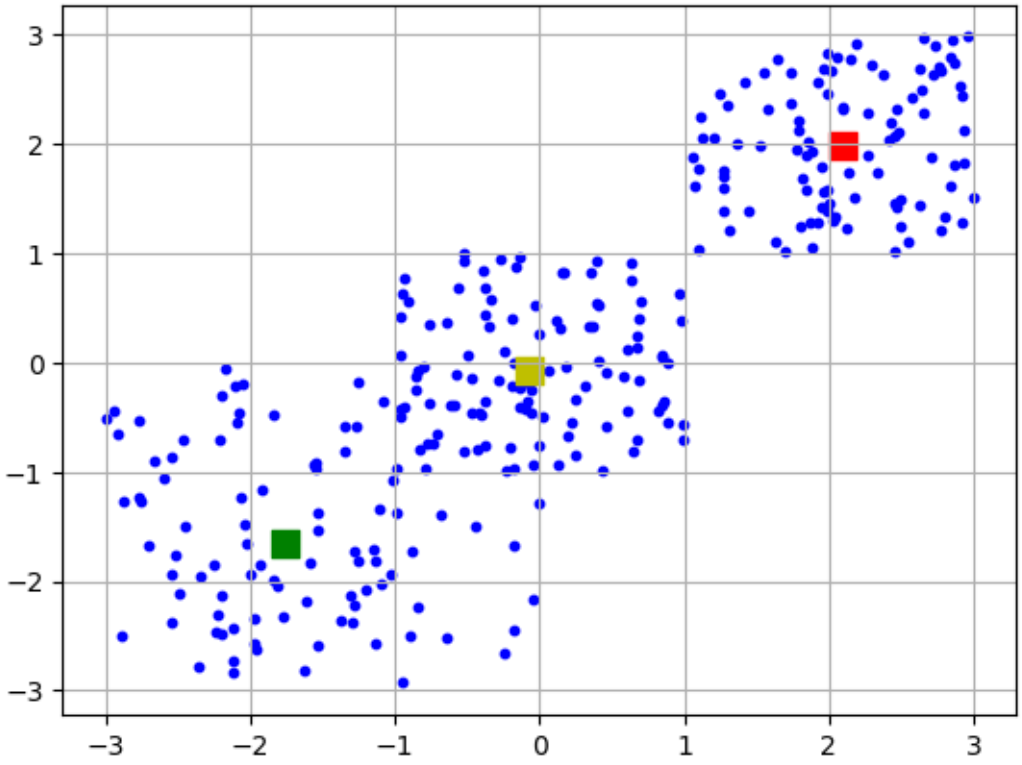
We can now generate a scatter plot using the data points stored in `X` and marks the cluster centers with large square markers of different colors.

- The first scatter plot command (`plt.scatter(X[:, 0], X[:, 1], s=10, c='b')`) plots the data points in `X` with blue markers of size 10.
- The subsequent scatter plot commands plot each cluster center individually (`plt.scatter(centers[i][0], centers[i][1], s=100, c=color, marker='s')`). Each cluster center is plotted with a square marker (`marker='s'`) of size 100 and a different color (`'r'`, `'g'`, and `'y'` for red, green, and yellow, respectively).

Finally, grid lines are added to the plot, and the plot is displayed. This visualization helps in understanding the distribution of data points and the positions of the cluster centers identified by the KMeans algorithm.

[7]:
```
# Scatter plot of the data points in X
# X[:, 0] selects the first column of X (x-coordinates)
# X[:, 1] selects the second column of X (y-coordinates)
# s = 10 sets the size of the markers to 10
# c = 'b' sets the color of the markers to blue
plt.scatter(X[:, 0], X[:, 1], s=10, c='b')

# Scatter plot of the cluster centers
# centers[0][0] and centers[0][1] represent the x and y coordinates of the␣
 ↪first cluster center
# s = 100 sets the size of the markers to 100
# c = 'r' sets the color of the markers to red
# marker='s' sets the marker shape to a square
plt.scatter(centers[0][0], centers[0][1], s=100, c='r', marker='s')

# Scatter plot of the second cluster center
# Similar parameters as above, but with different center coordinates and color
```

```
plt.scatter(centers[1][0], centers[1][1], s=100, c='g', marker='s')

# Scatter plot of the third cluster center
# Similar parameters as above, but with different center coordinates and color
plt.scatter(centers[2][0], centers[2][1], s=100, c='y', marker='s')

# Displaying grid lines on the plot
plt.grid()

# Showing the plot
plt.show()
```



Here is the code for getting the labels property of the K-means clustering example dataset; that is, how the data points are categorized into the tree clusters. Please note that in the context of clustering, an instance's label is the index of the cluster that this instance gets assigned to by the algorithm: **this is not to be confused with the class labels in classification** (remember that clustering is an unsupervised learning task). The KMeans instance preserves a copy of the labels of the instances it was trained on, available via the `labels_` instance variable:

[8]:
```
Kmean.labels_
```

[8]:
```
array([1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 2, 1, 2, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 2, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Let's use the code below for predicting the cluster of a data point:

[9]:
```
sample_test=np.array([1.5,1.5])
second_test=sample_test.reshape(1, -1)
Kmean.predict(second_test)
```

[9]:
```
array([0])
```

In the `KMeans` class, the transform() method measures the distance from each instance to every centroid:

[10]:
```
Kmean.transform(second_test)
```

[10]:
```
array([[0.77251091, 4.53737637, 2.22294236]])
```

How Find the Right Number of Clusters

- Determining the optimal number of clusters, often denoted as k, in K-means clustering is a crucial step in the process.
- While there's no definitive method to find the exact number of clusters, finding the right number of clusters in K-means clustering involves a combination of statistical techniques, domain expertise, and subjective judgment.
- It's essential to consider various methods and validate the results to ensure that the chosen number of clusters accurately captures the underlying patterns in the data.

Inertia or Within Cluster Sum of Squares (WCSS)

- A measure of the performance of the algorithm is the within cluster sum of squares also known as ***inertia***;
- For any given k the objective is to minimize inertia:

$$Inertia = \sum_{i=1}^{n} d_i^2 \tag{2}$$

where d_i is the distance of observation i from its cluster center

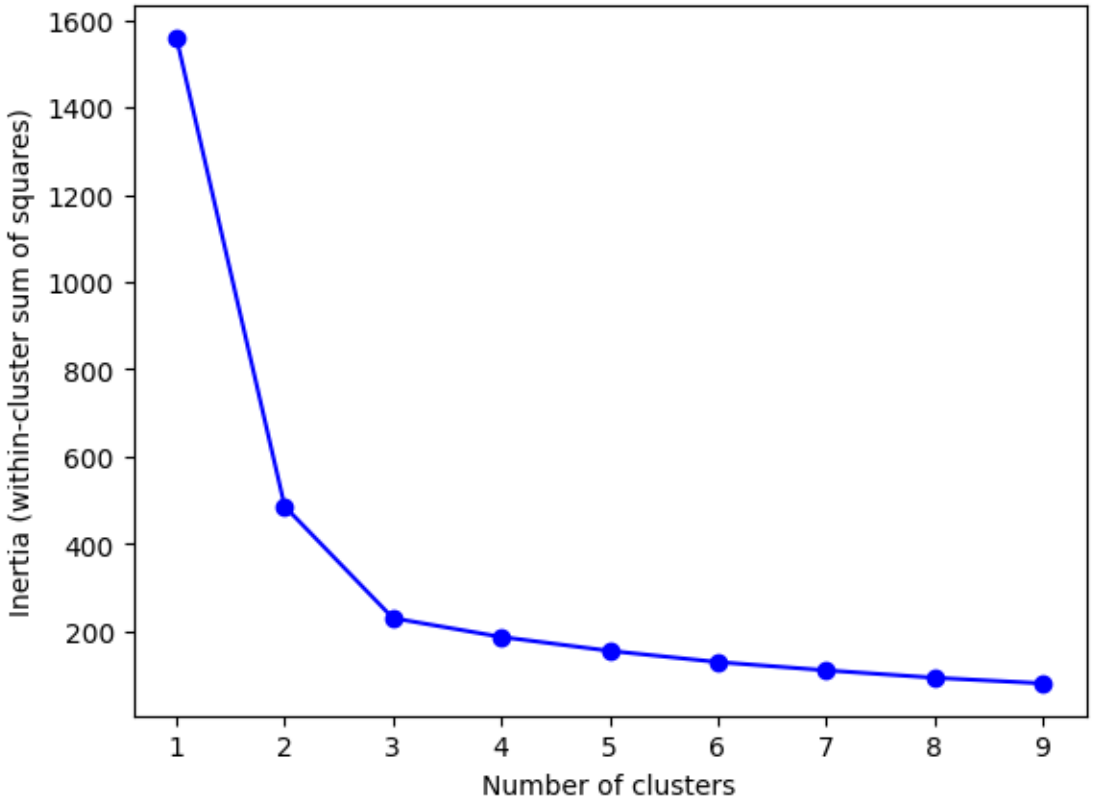
- In practice we use the k-means algorithm with several different starting points and choose the result that has the smallest inertia

You might be thinking that we could just pick the model with the lowest inertia, right? Unfortunately, it is not that simple. The inertia is not a good performance metric when trying to choose k since it keeps getting lower as we increase k. Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. Let's plot the inertia as a function of k

[11]:
```
Ks = range(1, 10)
inertia = [KMeans(i).fit(X).inertia_ for i in Ks]

fig = plt.figure()
```

```
<Figure size 640x480 with 0 Axes>
```

[12]:
```
plt.xlabel('Number of clusters')
plt.ylabel('Inertia (within-cluster sum of squares)')
plt.plot(Ks, inertia, '-bo')
plt.show()
```



As you can see, the inertia drops very quickly as we increase k up to 3, but then it decreases much more slowly as we keep increasing k. This curve has roughly the shape of an arm, and there is an "elbow" at $k = 3$ so if we did not know better, it would be a good choice: any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

Elbow Method

- Plot the within-cluster sum of squares (WCSS) against the number of clusters.
- WCSS measures the compactness of the clusters.
- The idea is to identify the "elbow" point in the plot, where the rate of decrease in WCSS slows down significantly. This point represents a good balance between minimizing intra-cluster distance and avoiding overfitting.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise approach (but also more computationally expensive) is to use the silhouette score, which is the mean silhouette coefficient over all the instances. An instance's silhouette coefficient is equal to $(b–a)/max(a, b)$ where a is the mean distance to the other instances in the same cluster (it is the mean intra-cluster distance), and b is the mean nearest-cluster distance, that is the mean distance to the instances of the next closest cluster (defined as the one that minimizes b, excluding the instance's own cluster).

Silhouette Analysis

For each observation i calculate $a(i)$, the average distance from other observations in its cluster, and $b(i)$, the average distance from observations in the closest other cluster. The silhouette score for observation i, $s(i)$, is defined as

$$s(i) = \frac{b(i) - a(i)}{\max[a(i), b(i)]} \tag{3}$$

The silhouette coefficient can vary between -1 and $+1$: a coefficient close to $+1$ means that the instance is **well inside its own cluster and far from other clusters**, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster. So we have to choose the number of clusters that **maximizes the average silhouette** score across all observations

1.3.3 Example 2 - Country Risk

The Country Risk Dataset (J. C. Hull, 2019, Chapter 2)

Consider the problem of understanding the risk of countries for foreign investment. Among the features that can be used for this are:

- GDP growth rate (IMF)
- Corruption index (Transparency international)
- Peace index (Institute for Economics and Peace)
- Legal Risk Index (Property Rights Association)

[36]:
```
# loading packages
```

```
import os

import pandas as pd
import numpy as np

# plotting packages
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as clrs

# Kmeans algorithm from scikit-learn
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
```

Values for each of the features for 122 countries are found in the `countryriskdata.csv` (available here)

[37]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[38]:
```
# load raw data
raw = pd.read_csv(os.path.join(path, 'countryriskdata.csv'))

# check the raw data
print("Size of the dataset (row, col): ", raw.shape)
print("\nFirst 5 rows\n", raw.head(n=5))
```

```
Size of the dataset (row, col):  (122, 6)

First 5 rows
      Country Abbrev  Corruption  Peace  Legal  GDP Growth
0    Albania     AL          39  1.867  3.822       3.403
1    Algeria     DZ          34  2.213  4.160       4.202
2  Argentina     AR          36  1.957  4.568      -2.298
3    Armenia     AM          33  2.218  4.126       0.208
4  Australia     AU          79  1.465  8.244       2.471
```

The GDP growth rate (%) is typically a positive or negative number with a magnitude less than 10. The corruption index is on a scale from 0 (highly corrupt) to 100 (no corruption). The peace index is on a scale from 1 (very peaceful) to 5 (not at all peaceful). The legal risk index runs from 0 to 10 (with high values being favorable).

Simple exploratory analysis: Print summary statistics

Note that all features have quite different variances, and Corruption and Legal are highly correlated.

[39]:
```
# print summary statistics
print("\nSummary statistics\n", raw.describe())
```

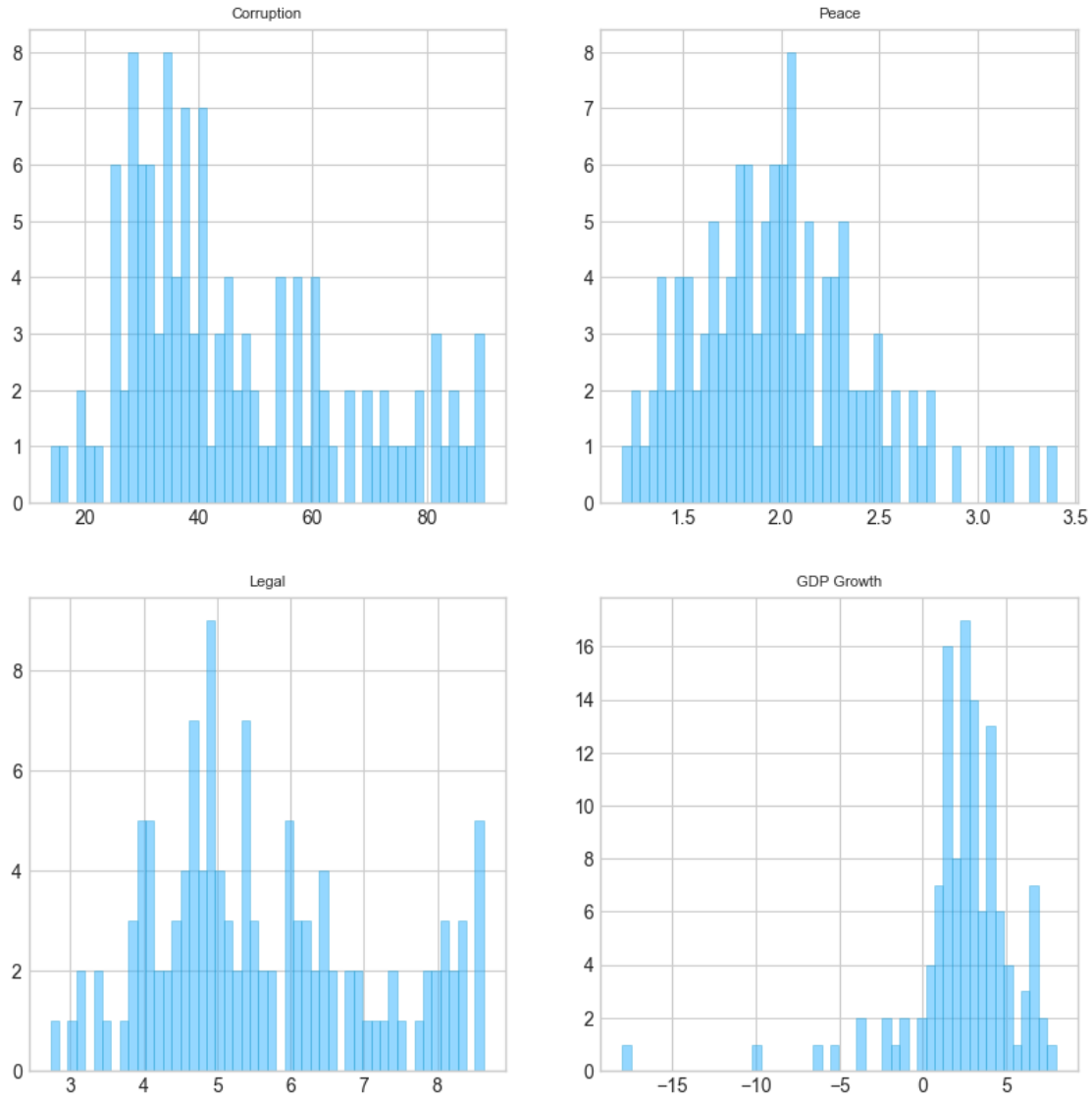
```
Summary statistics
        Corruption       Peace       Legal  GDP Growth
count  122.000000  122.000000  122.000000  122.000000
mean    46.237705    2.003730    5.598861    2.372566
std     19.126397    0.447826    1.487328    3.241424
min     14.000000    1.192000    2.728000  -18.000000
25%     31.250000    1.684750    4.571750    1.432250
50%     40.000000    1.969000    5.274000    2.496000
75%     58.750000    2.280500    6.476750    4.080000
max     90.000000    3.399000    8.633000    7.958000
```

Plot histogram

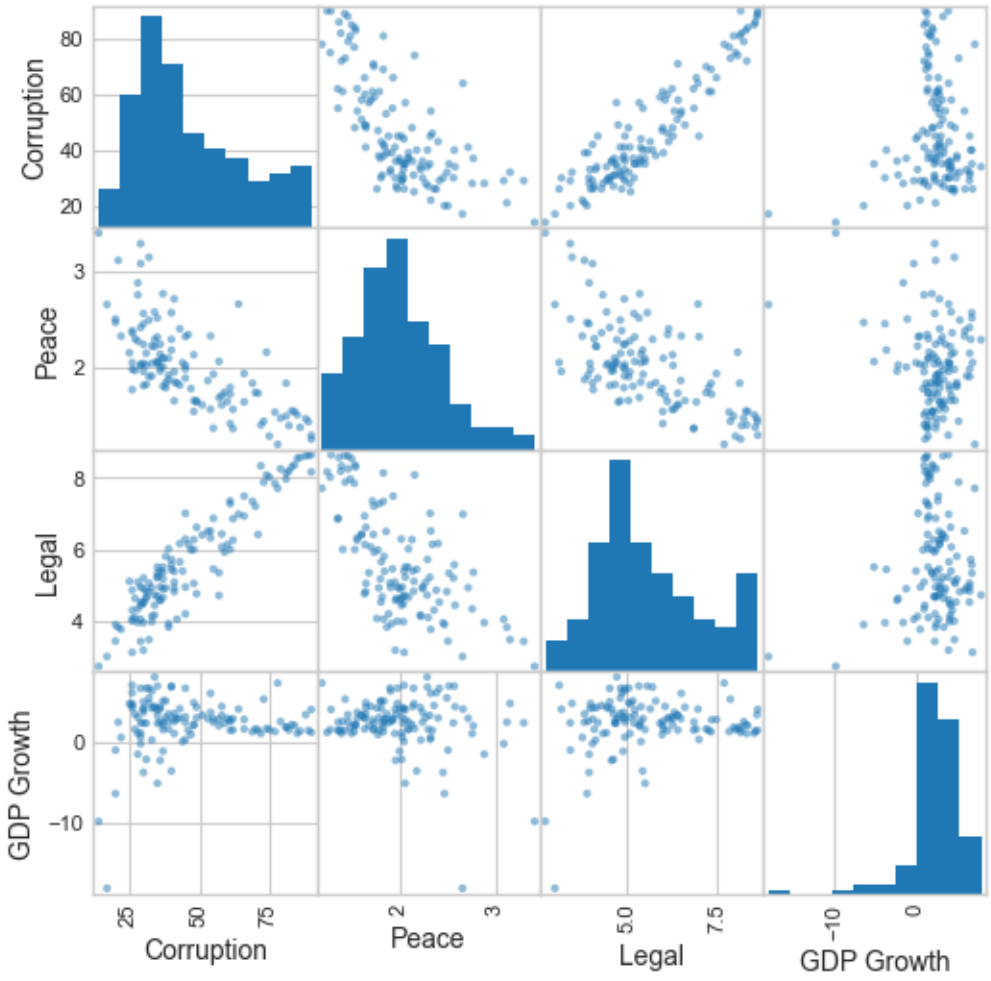
Note that distributions for GDP Growth is quite skewed.

[40]:
```
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
plt.style.use('seaborn-whitegrid') # nice and clean grid

# Plotting on the first subplot
axs[0, 0].hist(raw['Corruption'], alpha = 0.5, bins=50, facecolor = '#2ab0ff',␣
 ↪edgecolor='#169acf', linewidth=0.5)
axs[0, 0].set_title('Corruption', size = 8)

# Plotting on the first subplot
axs[0, 1].hist(raw['Peace'], alpha = 0.5, bins=50, facecolor = '#2ab0ff',␣
 ↪edgecolor='#169acf', linewidth=0.5)
axs[0, 1].set_title('Peace', size = 8)

# Plotting on the first subplot
axs[1, 0].hist(raw['Legal'], alpha = 0.5, bins=50, facecolor = '#2ab0ff',␣
 ↪edgecolor='#169acf', linewidth=0.5)
axs[1, 0].set_title('Legal', size = 8)

# Plotting on the first subplot
axs[1, 1].hist(raw['GDP Growth'], alpha = 0.5, bins=50, facecolor = '#2ab0ff',␣
 ↪edgecolor='#169acf', linewidth=0.5)
axs[1, 1].set_title('GDP Growth', size = 8)
```

[40]:
```
Text(0.5, 1.0, 'GDP Growth')
```



[41]:
```
# Creating a scatter plot matrix
pd.plotting.scatter_matrix(raw, figsize=(6, 6))
plt.show()
```



[42]:
```
# Calculate Pearson correlation (default)
pearson_corr = raw['Corruption'].corr(raw['Legal'])
print(f"Pearson correlation: {pearson_corr}")
```

```
Pearson correlation: 0.9235892197888613
```

Features Selection and Normalization

Since Corruption and Legal are highly correlated, we drop the Corruption variable, i.e., we pick three features for this analysis, Peace, Legal and GDP Grwoth. Let's normalize all the features, effectively making them equally weighted.

Ref. Feature normalization.

[43]:
```
X = raw[['Peace', 'Legal', 'GDP Growth']]
X = (X - X.mean()) / X.std()
print(X.head(5))
```

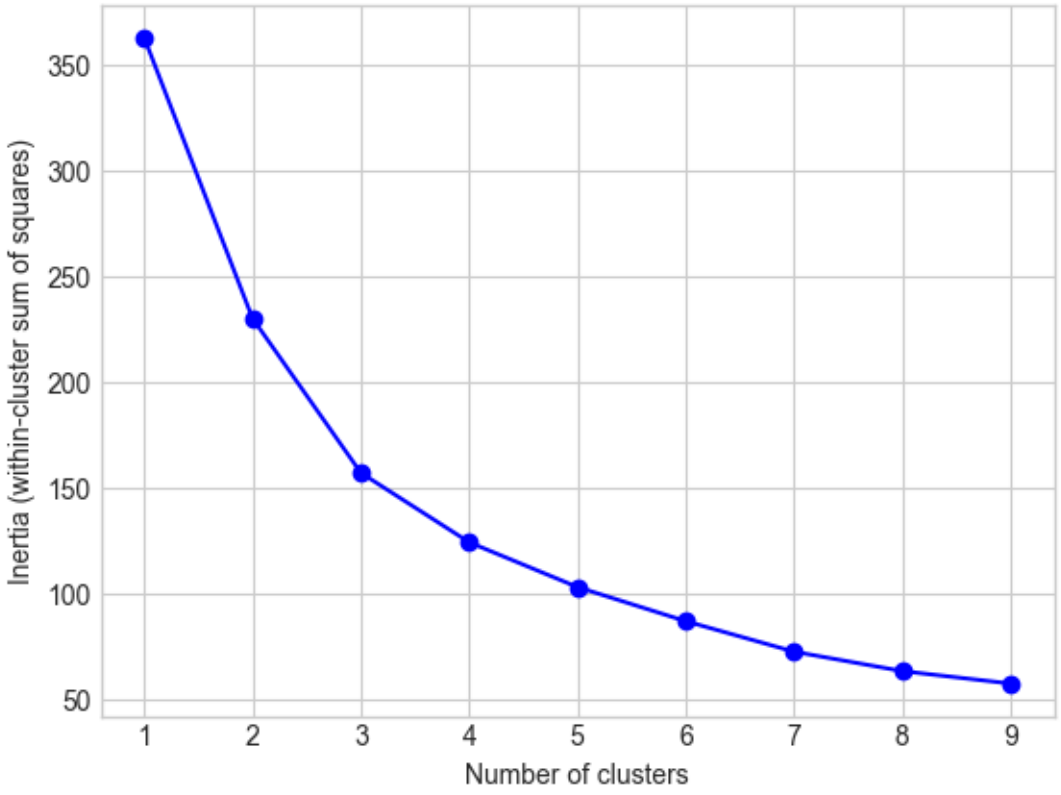
```
      Peace     Legal  GDP Growth
0 -0.305319 -1.194666    0.317896
1  0.467304 -0.967413    0.564392
2 -0.104348 -0.693096   -1.440899
3  0.478469 -0.990273   -0.667782
4 -1.202990  1.778450    0.030368
```

Perform Elbow Method

In our case, the marginal gain of adding one cluster dropped quite a bit from k=3 to k=4. We will choose k=3 (not a clear cut though).

Ref. Determining the number of clusters in a dataset.

[44]:
```
Ks = range(1, 10)
inertia = [KMeans(i).fit(X).inertia_ for i in Ks]

fig = plt.figure()
plt.plot(Ks, inertia, '-bo')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia (within-cluster sum of squares)')
plt.show()
```



Perform Silhouette Analysis

Next cell code performs silhouette analysis to determine the optimal number of clusters for KMeans clustering. Here's a description:

- **Range of Clusters**: The `range_n_clusters` list contains a range of candidate numbers of clusters to evaluate. This range is typically chosen based on domain knowledge or heuristic methods.
- **Silhouette Scores**: The `silhouette` list is initialized to store silhouette scores corresponding to each number of clusters.
- **Loop**: The code iterates through each candidate number of clusters.
- **KMeans Clustering**: For each iteration, a `KMeans` object is created with the specified number of clusters.
- **Clustering Data**: The `fit_predict` method is used to assign each data point to a cluster and obtain cluster labels.
- **Silhouette Score Calculation**: The `silhouette_score` function calculates the average silhouette score for the current clustering configuration.
- **Storing Scores**: The silhouette score is appended to the `silhouette` list.
- **Optional Printing**: Optionally, you can uncomment the print statement to display the silhouette score for each number of clusters.

By analyzing the silhouette scores for different numbers of clusters, you can determine the optimal number of clusters that best capture the underlying structure of the data.

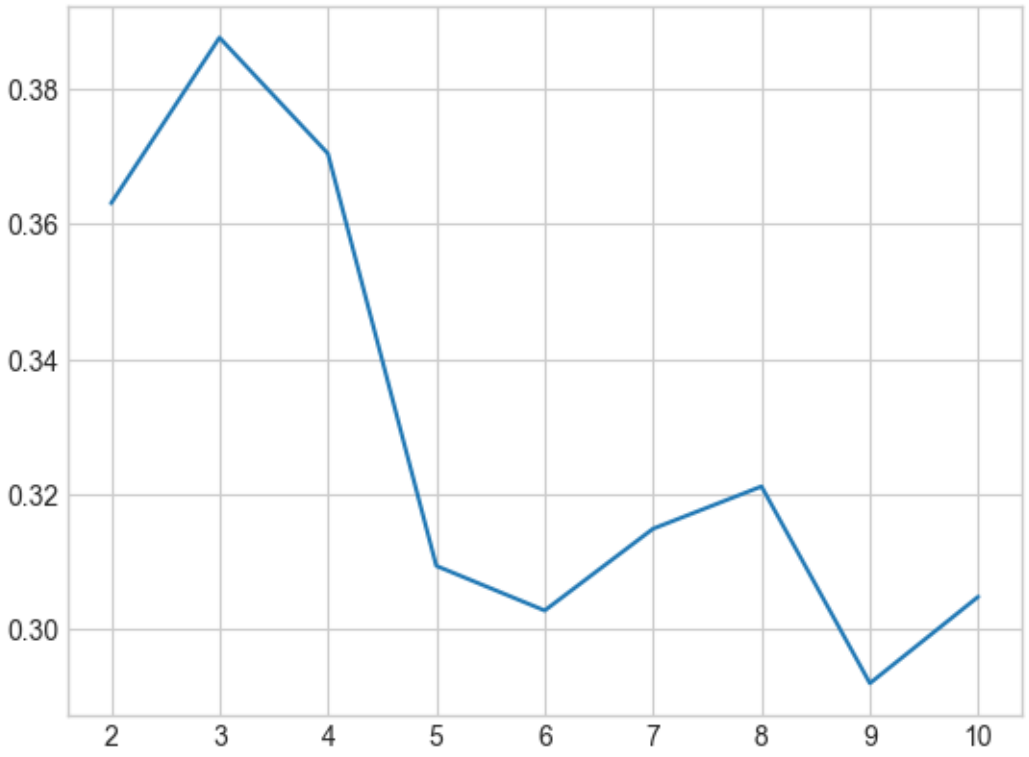
[45]:
```
# List of candidate numbers of clusters for silhouette analysis
range_n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10]

# List to store silhouette scores corresponding to each number of clusters
silhouette = []

# Iterating through each candidate number of clusters
for n_clusters in range_n_clusters:
    # Creating a KMeans object with a specific number of clusters
    clusterer = KMeans(n_clusters=n_clusters, random_state=0)
    # Assigning each data point to a cluster and obtaining cluster labels
    cluster_labels = clusterer.fit_predict(X)
    # Calculating the average silhouette score for the current clustering
    silhouette_avg = silhouette_score(X, cluster_labels)
    # Appending the silhouette score to the list
    silhouette.append(silhouette_avg)
    # Optional: Printing the silhouette score for each number of clusters
    print("For n_clusters=", n_clusters, "The average silhouette_score is :",␣
 ↪silhouette_avg)
```

```
For n_clusters= 2 The average silhouette_score is : 0.3630420703158315
For n_clusters= 3 The average silhouette_score is : 0.38757393707048954
```

```
For n_clusters= 4 The average silhouette_score is : 0.3704108671833623
For n_clusters= 5 The average silhouette_score is : 0.30937227721525684
For n_clusters= 6 The average silhouette_score is : 0.3027632280090134
For n_clusters= 7 The average silhouette_score is : 0.3148738253240533
For n_clusters= 8 The average silhouette_score is : 0.3211329731549249
For n_clusters= 9 The average silhouette_score is : 0.2919968985290435
For n_clusters= 10 The average silhouette_score is : 0.3048137513497579
```

[46]:
```
plt.plot(range_n_clusters, silhouette)
```

[46]:
```
[<matplotlib.lines.Line2D at 0x22500e10370>]
```



As you can see, this visualization is much richer than the previous one: in particular, although it confirms that $k = 3$ is a very good choice, it also underlines the fact that $k = 4$ is quite good as well, and much better than $k = 5$ or 7. This was not visible when comparing inertias.

k-means

[47]:
```
k = 3
kmeans = KMeans(n_clusters=k, random_state=0)
kmeans.fit(X)

# print inertia & cluster center
```

```
print("inertia for k={} is {}\n".format(k, kmeans.inertia_))
for i in range(k):
    print("cluster nr {} coordinates : {}".format(i, kmeans.
 ↪cluster_centers_[i]))

# take a quick look at the result
y = kmeans.labels_
print("\ncluster labels: ", y)
```

```
inertia for k=3 is 157.31661757321712

cluster nr 0 coordinates : [ 0.26568525 -0.45116779  0.36312086]
cluster nr 1 coordinates : [-0.96978306  1.17216616  0.00173193]
cluster nr 2 coordinates : [ 1.3920898  -1.04170733 -1.79449174]

cluster labels:  [0 0 2 0 1 1 2 0 0 1 0 0 0 1 2 0 2 0 1 2 1 0 0 1 0 0 1 2 1 0 2
0 0 1 0 1 1
 0 0 1 0 0 0 0 1 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 2 0 1 0 0 1 0 0 1 0 0 0 0 0 0
 0 1 1 0 2 1 0 0 0 0 0 0 1 1 1 0 2 0 0 0 0 0 1 1 1 0 1 0 1 1 1 0 0 0 2 0 0
 0 2 1 1 1 1 2 0 2 0 0]
```

Visualize the result (3D plot)

> **NOTE** In Matplotlib, a colormap (colormap object or cmap) is a mapping from data values to colors. It is used to visually represent scalar data in plots such as scatter plots, surface plots, and heatmaps. The choice of colormap can significantly impact the interpretation and readability of the plot.
>
> Matplotlib provides various built-in colormaps, each with its own characteristics, allowing users to choose the one that best suits their data and visualization needs. Some commonly used colormaps include 'viridis', 'plasma', 'inferno', 'magma', 'cividis', 'coolwarm', 'hot', 'jet', 'rainbow', and many others.
>
> The `matplotlib.cm` module contains functions and classes related to colormaps. One of the main functions is `matplotlib.cm.get_cmap(name, lut=None)`, which retrieves a colormap by name. This function returns a colormap object that can be used to map scalar data to colors.
>
> Additionally, you can directly access specific colormaps by name without importing the `matplotlib.cm` module. For example, `cm.viridis` accesses the 'viridis' colormap, `cm.plasma` accesses the 'plasma' colormap, and so on.
>
> You can use colormaps in various plotting functions by specifying the colormap argument (`cmap`). For example, in a scatter plot, you can set `cmap='viridis'` to use the 'viridis' colormap.
>
> Colormaps are particularly useful for enhancing the visualization of data and conveying additional information through colors. However, it's essential to choose colormaps carefully to ensure that they are perceptually uniform and suitable for the specific data being visualized.

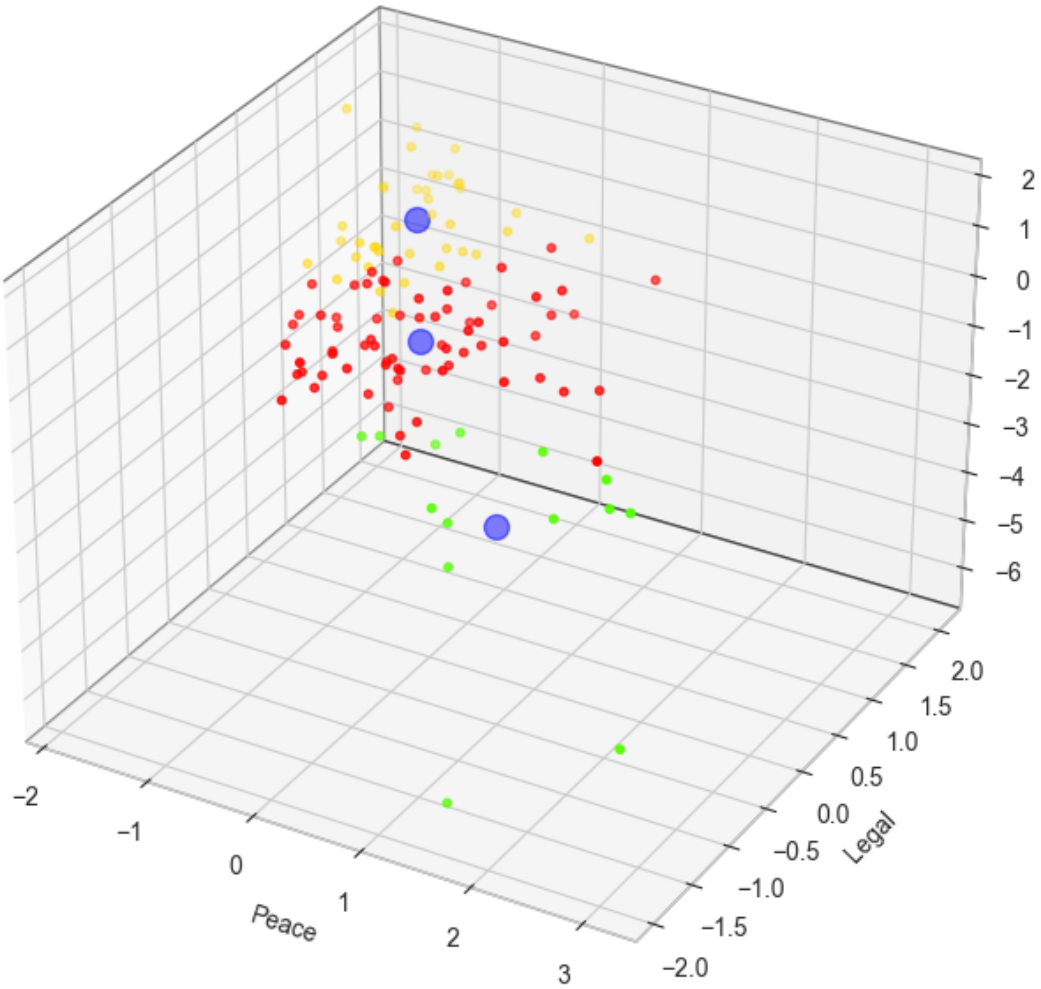
NOTE The 'prism' colormap in Matplotlib is a perceptually uniform colormap designed to smoothly transition through the visible spectrum. It creates a rainbow-like effect where colors shift continuously from one end of the spectrum to the other.

In the 'prism' colormap:

Low values are represented by violet or blue hues. Intermediate values are represented by a mix of colors ranging from blue to green, yellow, and orange. High values are represented by red hues.

[48]:
```
#
# Set up the color. The matplotlib.colors.Normalize class belongs to the␣
 ↪matplotlib.colors module.
# The matplotlib.colors module is used for converting color or numbers␣
 ↪arguments to RGBA or RGB.
# This module is used for mapping numbers to colors or color specification␣
 ↪conversion in a 1-D
# array of colors also known as colormap. The matplotlib.colors.Normalize class␣
 ↪is used to normalize
# data into the interval of [0.0, 1.0].
#
norm = clrs.Normalize(vmin=0.,vmax=y.max())
#
# See https://matplotlib.org/stable/tutorials/colors/colormaps.html for␣
 ↪choosing the appropriate color
# map for 3-D plot
#
# Setting colormap to 'prism' from the matplotlib.cm module
cmap = cm.prism

# Creating a figure object with a 3D subplot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Setting edge color of the axes to black
ax.xaxis.pane.set_edgecolor('k')
ax.yaxis.pane.set_edgecolor('k')
ax.zaxis.pane.set_edgecolor('k')

# Scatter plotting the data points in 3D space
ax.scatter(X.iloc[:,0], X.iloc[:,1], X.iloc[:,2], c=cmap(norm(y)), marker='o',␣
 ↪s=10)

# Retrieving and plotting cluster centers
centers = kmeans.cluster_centers_
for i in range(k):
    ax.scatter(centers[i, 0], centers[i, 1], centers[i, 2], c='blue', s=100,␣
 ↪alpha=0.5, marker='o')
```

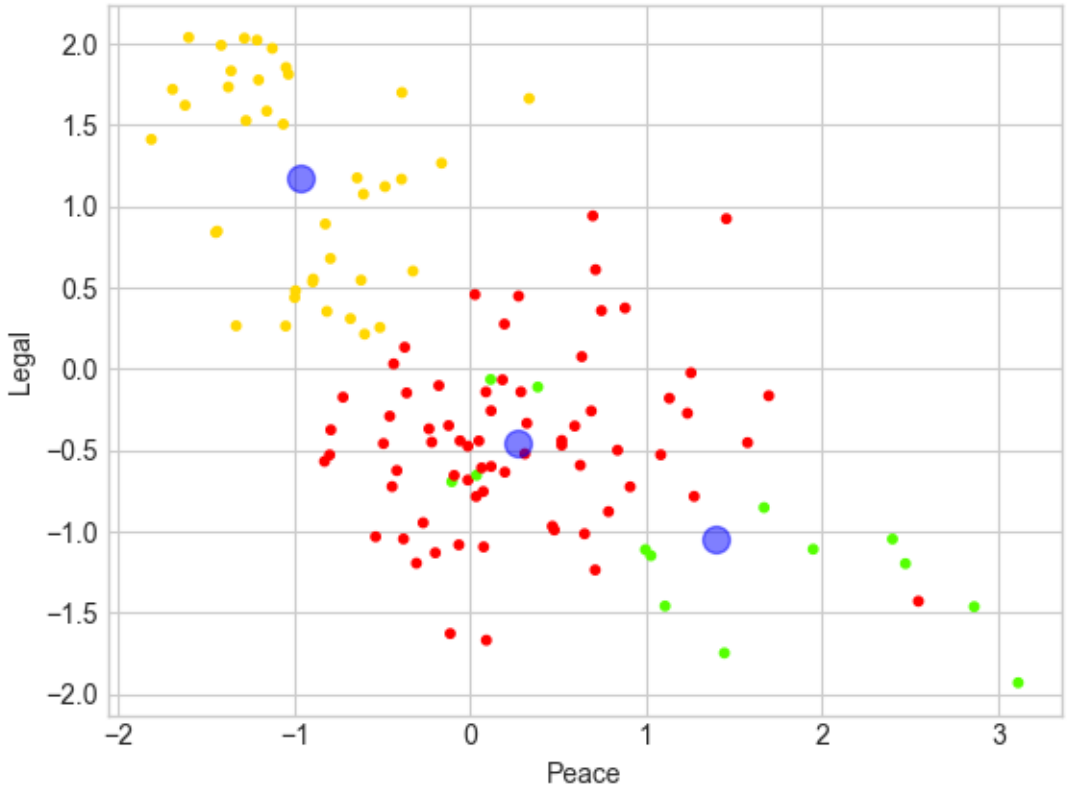
```
# Setting labels for each axis
ax.set_xlabel('Peace')
ax.set_ylabel('Legal')
ax.set_zlabel('GDP Growth')

# Displaying the plot
plt.show()
```

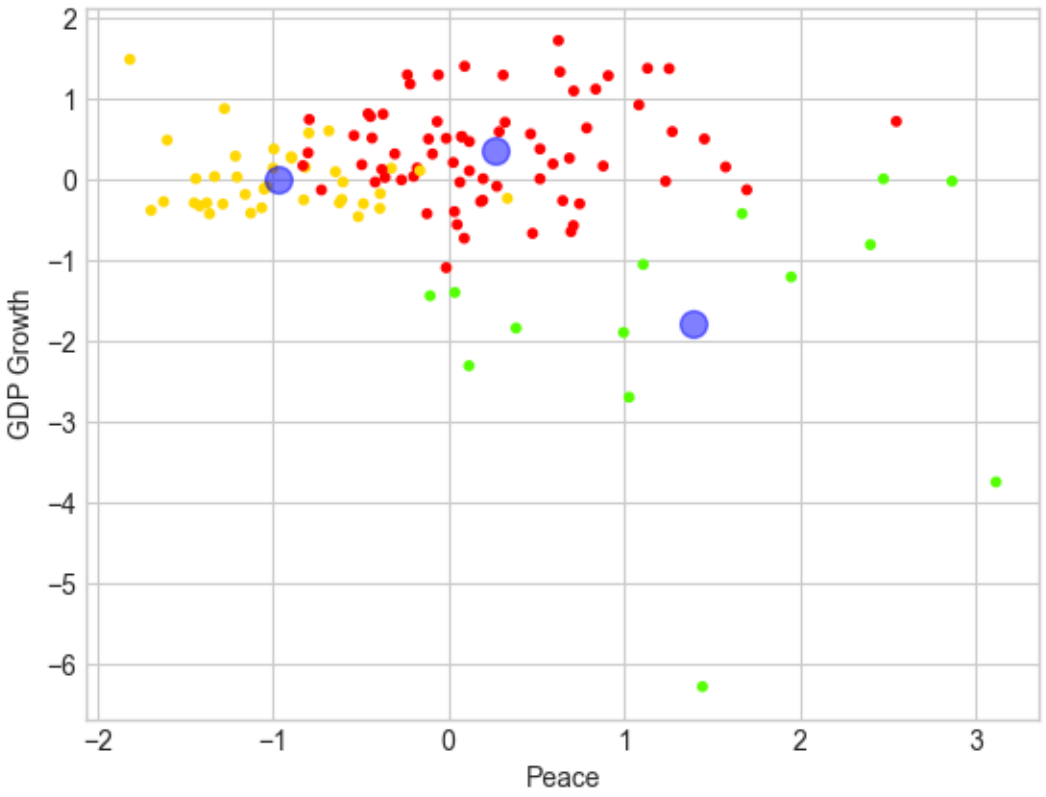


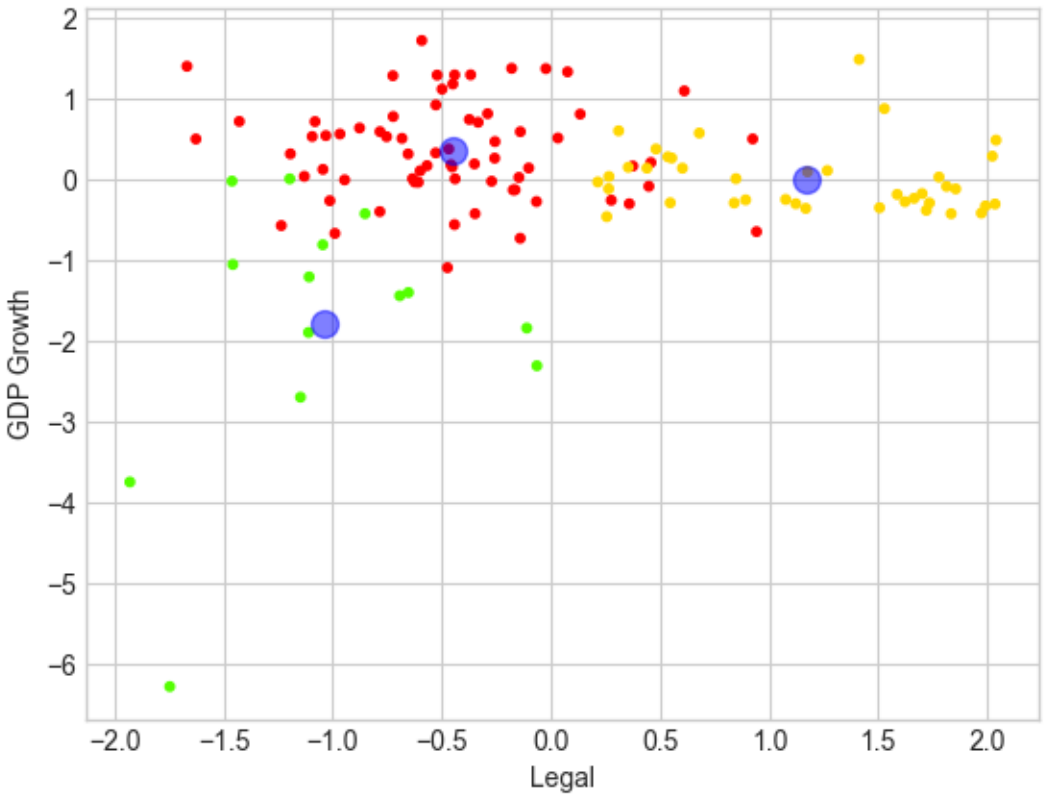
Visualize the result (3 2D plots)

[49]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
figs = [(0, 1), (0, 2), (1, 2)]
labels = ['Peace', 'Legal', 'GDP Growth']

for i in range(k):
    fig = plt.figure(i)
    plt.scatter(X.iloc[:,figs[i][0]], X.iloc[:,figs[i][1]], c=cmap(norm(y)),␣
 ↪s=10)
    plt.scatter(centers[:, figs[i][0]], centers[:, figs[i][1]], c='blue',␣
 ↪s=100, alpha=0.5)
    plt.xlabel(labels[figs[i][0]])
    plt.ylabel(labels[figs[i][1]])

plt.show()
```




GDP Growth
Peace
2
1
0
−1
−2
−3
−4
−5
−6
−2
−1
0
1
2
3

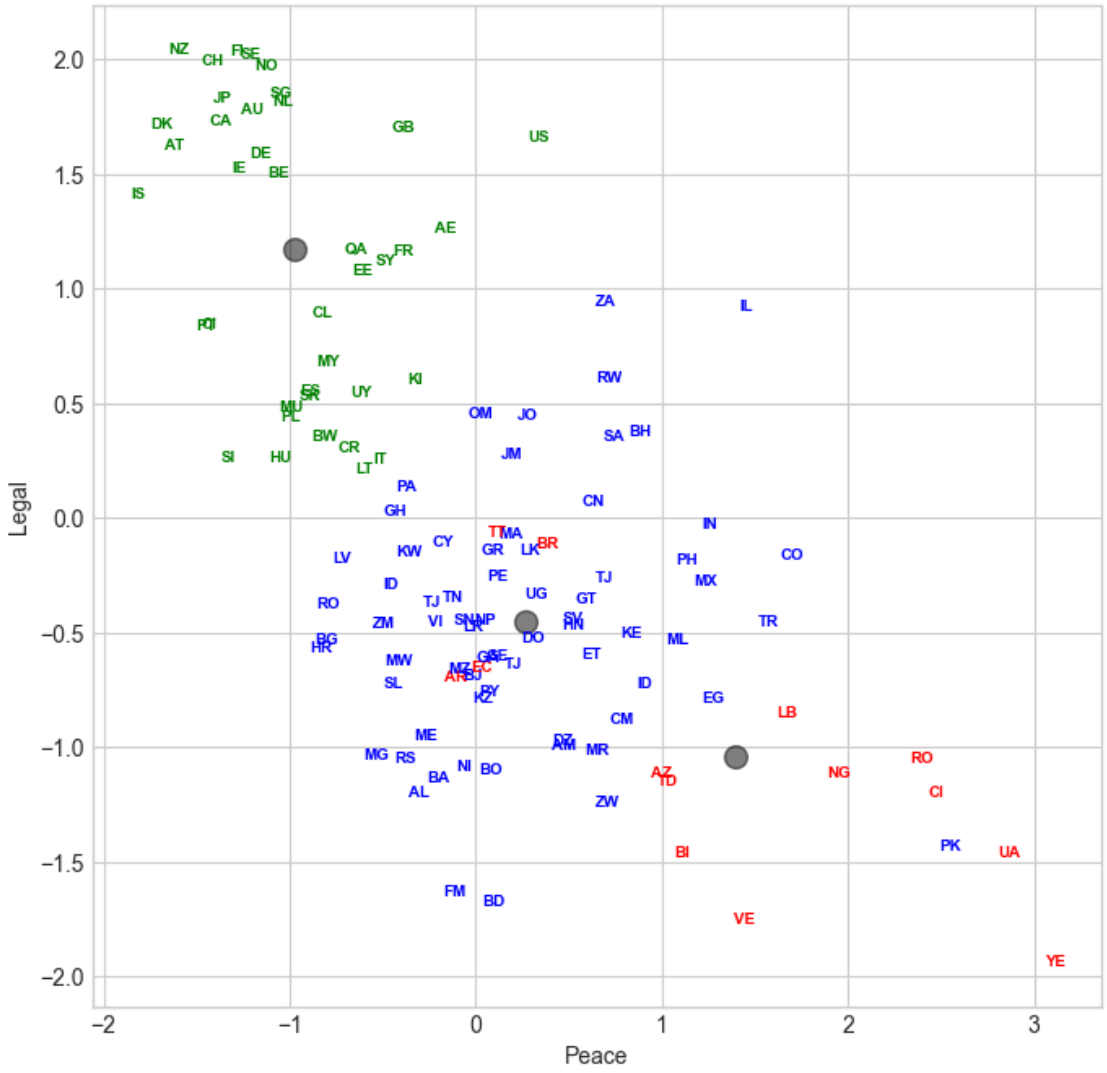


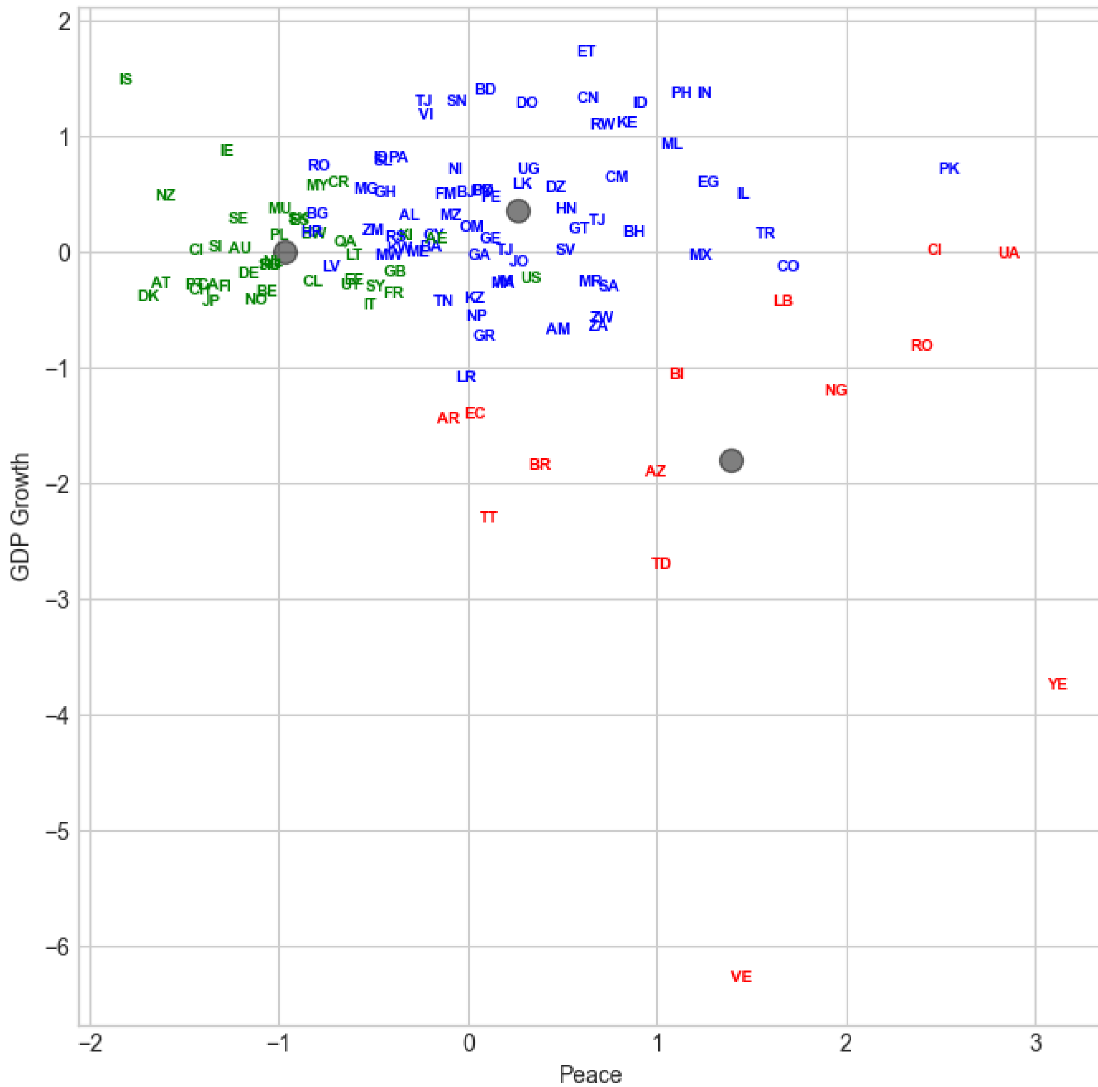
Visualize the result (3 2D plots)

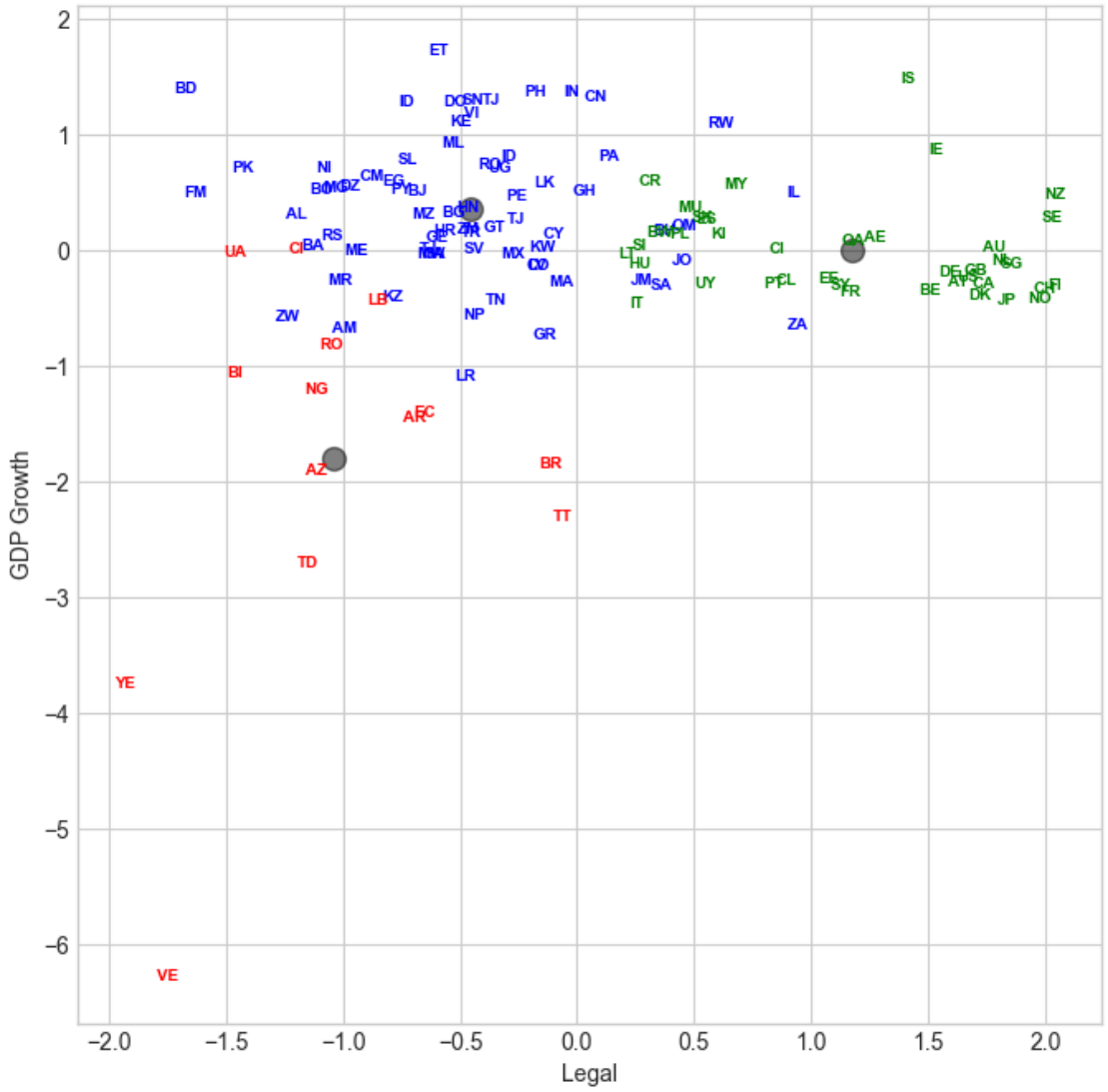
plot country abbreviations instead of dots.

[50]:
```
%matplotlib inline
import matplotlib.pyplot as plt

figs = [(0, 1), (0, 2), (1, 2)]
labels = ['Peace', 'Legal', 'GDP Growth']
colors = ['blue','green', 'red']

for i in range(3):
    fig = plt.figure(i, figsize=(8, 8))
    x_1 = figs[i][0]
    x_2 = figs[i][1]
    plt.scatter(X.iloc[:, x_1], X.iloc[:, x_2], c=y, s=0, alpha=0)
    plt.scatter(centers[:, x_1], centers[:, x_2], c='black', s=100, alpha=0.5)
    for j in range(X.shape[0]):
        plt.text(X.iloc[j, x_1], X.iloc[j, x_2], raw['Abbrev'].iloc[j],
                 color=colors[y[j]], weight='semibold', horizontalalignment =␣
 ↪'center', verticalalignment = 'center', size=7)
    plt.xlabel(labels[x_1])
```

```
    plt.ylabel(labels[x_2])

plt.show()
```




GDP Growth
Peace
2
1
0
−1
−2
−3
−4
−5
−6
−2
−1
0
1
2
3



List the result

[51]:
```
result = pd.DataFrame({'Country':raw['Country'], 'Abbrev':raw['Abbrev'],␣
  ↪'Label':y})
with pd.option_context('display.max_rows', None, 'display.max_columns', 3):
    print(result.sort_values('Label'))
```

```
                    Country Abbrev  Label
0                   Albania     AL      0
73                Mozambique     MZ      0
72                   Morocco     MA      0
71                Montenegro     ME      0
70                   Moldova     FM      0
69                    Mexico     MX      0
```

67	Mauritania	MR	0
66	Mali	ML	0
64	Malawi	MW	0
63	Madagascar	MG	0
61	Liberia	LR	0
120	Zambia	ZM	0
59	Latvia	LV	0
58	Kuwait	KW	0
56	Kenya	KE	0
55	Kazakhstan	KZ	0
54	Jordan	JO	0
52	Jamaica	JM	0
50	Israel	IL	0
48	Iran	ID	0
74	Nepal	NP	0
47	Indonesia	ID	0
77	Nicaragua	NI	0
81	Pakistan	PK	0
118	Vietnam	VI	0
111	Uganda	UG	0
110	Turkey	TR	0
109	Tunisia	TN	0
107	The FYR of Macedonia	TJ	0
106	Thailand	TJ	0
105	Tanzania	TJ	0
101	Sri Lanka	LK	0
99	South Africa	ZA	0
95	Sierra Leone	SL	0
94	Serbia	RS	0
93	Senegal	SN	0
92	Saudi Arabia	SA	0
91	Rwanda	RW	0
89	Romania	RO	0
85	Philippines	PH	0
84	Peru	PE	0
83	Paraguay	PY	0
82	Panama	PA	0
80	Oman	OM	0
46	India	IN	0
121	Zimbabwe	ZW	0
21	China	CN	0
8	Bangladesh	BD	0
29	Dominican Republic	DO	0
22	Colombia	CO	0
10	Benin	BJ	0
31	Egypt	EG	0
32	El Salvador	SV	0
7	Bahrain	BH	0

11	Bolivia	BO	0
34	Ethiopia	ET	0
25	Cyprus	CY	0
3	Armenia	AM	0
24	Croatia	HR	0
37	Gabon	GA	0
38	Georgia	GE	0
15	Bulgaria	BG	0
40	Ghana	GH	0
17	Cameroon	CM	0
41	Greece	GR	0
1	Algeria	DZ	0
42	Guatemala	GT	0
43	Honduras	HN	0
12	Bosnia and Herzegovina	BA	0
13	Botswana	BW	1
96	Singapore	SG	1
97	Slovakia	SK	1
98	Slovenia	SI	1
44	Hungary	HU	1
103	Switzerland	CH	1
100	Spain	ES	1
102	Sweden	SE	1
88	Qatar	QA	1
104	Taiwan	SY	1
5	Austria	AT	1
4	Australia	AU	1
113	United Arab Emirates	AE	1
114	United Kingdom	GB	1
115	United States	US	1
116	Uruguay	UY	1
9	Belgium	BE	1
87	Portugal	PT	1
18	Canada	CA	1
65	Malaysia	MY	1
33	Estonia	EE	1
57	Korea (South)	KI	1
62	Lithuania	LT	1
86	Poland	PL	1
39	Germany	DE	1
68	Mauritius	MU	1
28	Denmark	DK	1
53	Japan	JP	1
35	Finland	FI	1
26	Czech Republic	CI	1
75	Netherlands	NL	1
76	New Zealand	NZ	1
51	Italy	IT	1

79	Norway	NO	1
20	Chile	CL	1
49	Ireland	IE	1
45	Iceland	IS	1
23	Costa Rica	CR	1
36	France	FR	1
117	Venezuela	VE	2
119	Yemen	YE	2
112	Ukraine	UA	2
16	Burundi	BI	2
108	Trinidad and Tobago	TT	2
6	Azerbaijan	AZ	2
30	Ecuador	EC	2
27	Democratic Republic of Congo	CI	2
78	Nigeria	NG	2
19	Chad	TD	2
14	Brazil	BR	2
90	Russia	RO	2
2	Argentina	AR	2
60	Lebanon	LB	2

1.3.4 Limits of K-Means

- Despite its many merits, most notably being fast and scalable, K-Means is not perfect.
- As we saw, it is necessary to run the algorithm several times to avoid sub-optimal solutions, plus you need to specify the number of clusters, which can be quite a hassle.
- Moreover, K-Means does not behave very well when the clusters have varying sizes, different densities, or non-spherical shapes.

Source: Géron (see References and Credits Section)

1.4 Hierarchical Clustering

Definition

- Hierarchical clustering algorithms are a family of algorithms used in machine learning for grouping data points into a hierarchy of clusters.
- These algorithms build a hierarchy of clusters either by starting with individual data points and successively merging them (**agglomerative approach**) or by beginning with the entire data set and successively dividing it into smaller clusters (**divisive approach**).
- The process creates a tree-like structure known as a **dendrogram**, which illustrates how individual elements are grouped into clusters.

Key Characteristics of Hierarchical Clustering:

1. **Agglomerative Approach**: This is the more commonly used method in hierarchical clustering. It starts by treating each data point as a single cluster. Then, iteratively, the closest

pairs of clusters are merged, creating a hierarchy from the bottom up. The process continues until all points are merged into a single cluster, or a stopping criterion is met.

2. **Divisive Approach**: In contrast, divisive methods start with the entire dataset as one large cluster and then partition it into smaller clusters. This partitioning continues recursively, leading to finer and more specific clusters, until each data point is in its own cluster or a stopping criterion is reached.

Source: https://i1.wp.com/rposts.com/wpcontent/uploads/2017/12/Agnes.png

Creating a Dendrogram

Both approaches produce a dendrogram, a tree-like diagram that records the sequences of merges or splits. The height at which two clusters are joined in the dendrogram represents the distance between these clusters, providing valuable insights into the data structure.

Distance Metrics and Linkage Criteria

- The choice of distance metric (e.g., Euclidean, Manhattan) and linkage criterion (e.g., single linkage, complete linkage, average linkage, see below for a description) plays a crucial role in hierarchical clustering. These determine how the similarity between clusters is calculated and directly influence the clustering outcome.
- The linkage criterion determines the distance between sets of observations as a function of the pairwise distances between observations. It's a rule used to decide when to combine two clusters into a single cluster as the algorithm progresses either bottom-up (agglomerative) or top-down (divisive). Here are the most common types of linkage criteria:

1. **Single Linkage (Nearest Point Algorithm)**: The distance between two clusters is defined as the shortest distance between any single member of one cluster and any single member of the other cluster. This can result in a chaining effect, where clusters end up being long and straggly.

Source: https://www.saedsayad.com/images/Clustering_single.png

2. **Complete Linkage (Farthest Point Algorithm)**: The distance between two clusters is defined as the longest distance between any single member of one cluster and any single member of the other cluster. This tends to produce more compact, tightly bound clusters.

Source:- https://www.saedsayad.com/images/Clustering_complete.png

3. **Average Linkage (Average of Pairs)**: The distance between two clusters is computed as the average distance between all pairs of individuals in the two clusters. This creates a balance between the single and complete linkage criteria and is less affected by outliers than either.

Source:- https://www.saedsayad.com/images/Clustering_average.png

Other Linkage Methods

- **Centroid Linkage (Minimum Centroid Distance)**: The distance between two clusters is the distance between their centroids - that is, the mean point of each cluster. Clusters are merged based on the proximity of their centroids.

- **Ward's Method (Minimum Variance Method)**: This approach aims to minimize the total within-cluster variance. At each step, the pair of clusters with the minimum between-cluster distance are merged. This method tends to create clusters of roughly equal size, which is useful in some contexts but not all.

The choice of linkage criterion can significantly affect the shape and the size of the clusters formed by hierarchical clustering, and it should be chosen based on the nature of the data and the desired clustering resolution.

The Distance Matrix

- The distance matrix, also known as a dissimilarity matrix, is a table that shows the distance between each pair of objects in a dataset. Each element (i, j) in the matrix represents the distance between the ith and jth elements of the dataset. The choice of distance metric (like Euclidean, Manhattan, etc.) depends on the type of data and the specific requirements of the analysis.
- For a dataset with N objects, the distance matrix is an $N \times N$ table where the diagonal elements are zeros because the distance between any object and itself is zero. The matrix is symmetric because the distance from object i to object j is the same as from object j to object i.
- When used in hierarchical clustering, this matrix serves as a fundamental component in the algorithm's iterative process of clustering objects.
- At each step, the algorithm consults the distance matrix to determine which objects or clusters are closest to each other and should be merged.
- After each merging step, the distance matrix is updated to reflect the distances between the new clusters and the remaining objects or clusters.
- The distance matrix is crucial because it encapsulates the entire pairwise dissimilarities within the dataset, allowing the hierarchical clustering algorithm to systematically organize the data into a dendrogram, which visually represents the cluster hierarchy.
- Let's see an example of computation of a Distance Matrix...

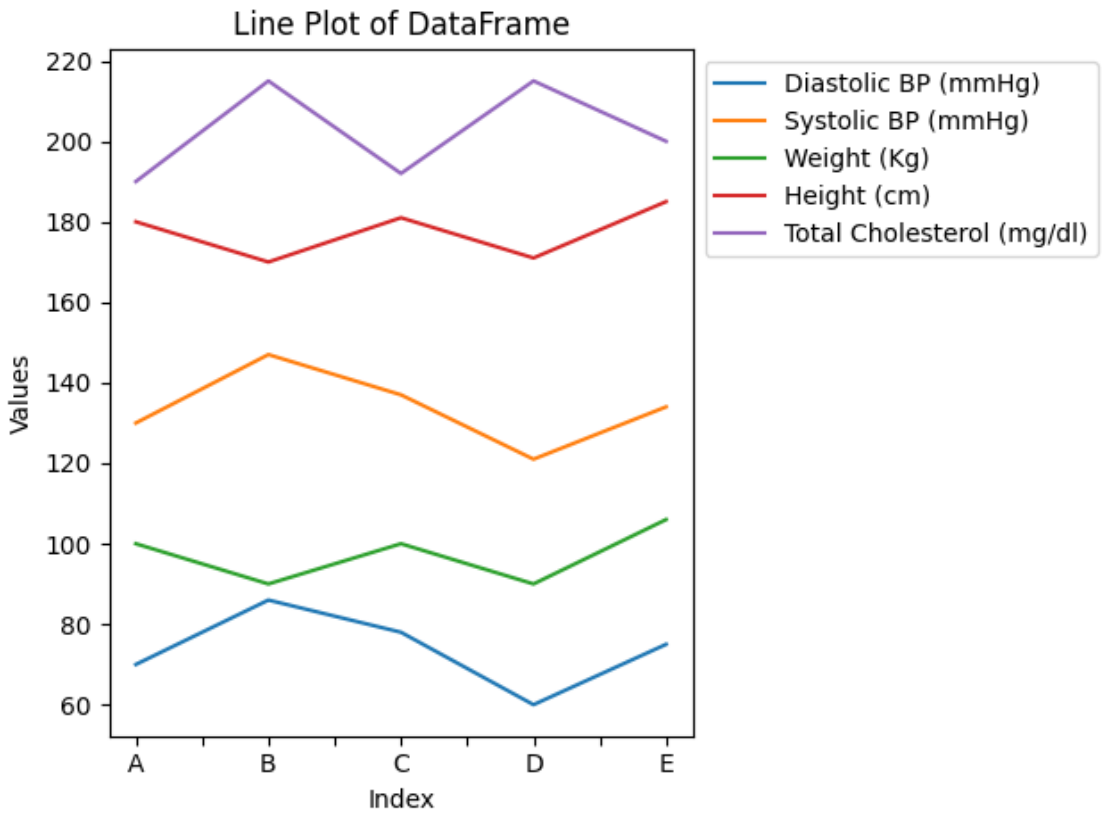
Example of Computation of a Distance Matrix

[13]:
```
import pandas as pd
import numpy as np

variables = ['A', 'B', 'C', 'D','E']
labels    = ['Diastolic BP (mmHg)', 'Systolic BP (mmHg)', 'Weight (Kg)',␣
 ↪'Height (cm)', 'Total Cholesterol (mg/dl)']
data      =␣
 ↪[[70,86,78,60,75],[130,147,137,121,134],[100,90,100,90,106],[180,170,181,171,185],[190,215,

df = pd.DataFrame(data, columns=variables, index=labels)
df
```

[13]:

	A	B	C	D	E
Diastolic BP (mmHg)	70	86	78	60	75
Systolic BP (mmHg)	130	147	137	121	134
Weight (Kg)	100	90	100	90	106
Height (cm)	180	170	181	171	185
Total Cholesterol (mg/dl)	190	215	192	215	200

[14]:
```
# Plotting the DataFrame using the default line plot
dft1 = df.T
dft1.plot()
plt.title('Line Plot of DataFrame')
plt.xlabel('Index')
plt.ylabel('Values')
# Place the legend outside the plot area to the right
# "loc" aligns the legend's upper left corner with the upper right corner of␣
 ↪the plot area
# "bbox_to_anchor" specifies the coordinates (relative to the plot area) where␣
 ↪the legend should be placed
plt.legend(loc='upper left', bbox_to_anchor=(1,1))
plt.tight_layout() # Adjust the layout to make room for the legend
plt.show()
```



[15]:
```
# Import pdist and squareform functions from scipy.spatial.distance to␣
 ↪calculate pairwise distances between points.
from scipy.spatial.distance import pdist, squareform

# Computing the distance matrix
distance_matrix = pd.DataFrame(
    # squareform converts a vector-form distance vector to a square-form␣
 ↪distance matrix,
    # and pdist computes the pairwise distances between observations in␣
 ↪n-dimensional space.
    squareform(pdist(df, metric='euclidean')),
    # Setting the columns and index of the resulting DataFrame to the index of␣
 ↪'df'
    # to maintain the same ordering and labeling.
    columns=df.index,
    index=df.index
)

# Displaying the computed distance matrix
distance_matrix
```

[15]:

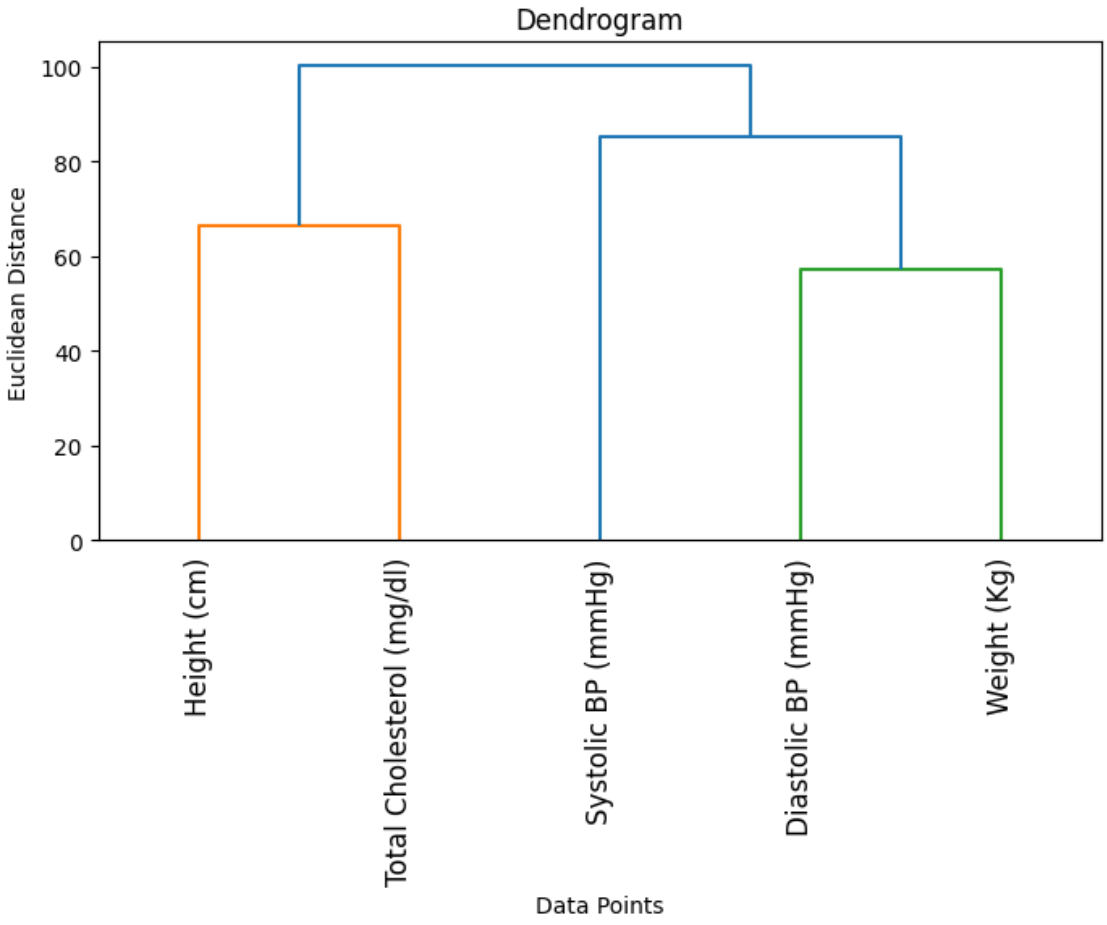
	Diastolic BP (mmHg)	Systolic BP (mmHg)	\
Diastolic BP (mmHg)	0.000000	134.178985	
Systolic BP (mmHg)	134.178985	0.000000	
Weight (Kg)	57.105166	85.223236	
Height (cm)	232.778865	100.329457	
Total Cholesterol (mg/dl)	289.287055	156.336176	

	Weight (Kg)	Height (cm)	Total Cholesterol (mg/dl)
Diastolic BP (mmHg)	57.105166	232.778865	289.287055
Systolic BP (mmHg)	85.223236	100.329457	156.336176
Weight (Kg)	0.000000	179.340458	238.012605
Height (cm)	179.340458	0.000000	66.385239
Total Cholesterol (mg/dl)	238.012605	66.385239	0.000000

[55]:
```
#from sklearn.cluster import AgglomerativeClustering

# Performing hierarchical clustering
#hc    = AgglomerativeClustering(n_clusters=3, affinity='euclidean',␣
 ↪linkage='single')
#y_hc = hc.fit_predict(data)
```

[16]:
```
from scipy.cluster.hierarchy import dendrogram, linkage

# Generating the linkage matrix
Z = linkage(data, 'single')
# Plotting the dendrogram
```

```
plt.figure(figsize=(8, 4))
dendrogram(Z, labels=labels, leaf_rotation=90, leaf_font_size=12)
plt.title("Dendrogram")
plt.xlabel("Data Points")
plt.ylabel("Euclidean Distance")
plt.show()
```



Data Normalization

- In addition to different linkage functions, it is important to select an appropriate distance metric that reflects how we define similarity.
- In the previous example, we grouped for example the variables body weight and the diastolic blood pressure because they had similar values.
- However, it might make more sense to group the diastolic blood pressure with the systolic blood pressure because these two variables had similar patterns across the individuals.
- If one individual has a high systolic blood pressure, the same individual is likely to also have a high diastolic blood pressure. The same is also true for the body weight and body height. One

way to group the variables based on similarity in their pattern or profile is to first standardize the variables so that all variables had mean of 0 and standard deviation of 1.

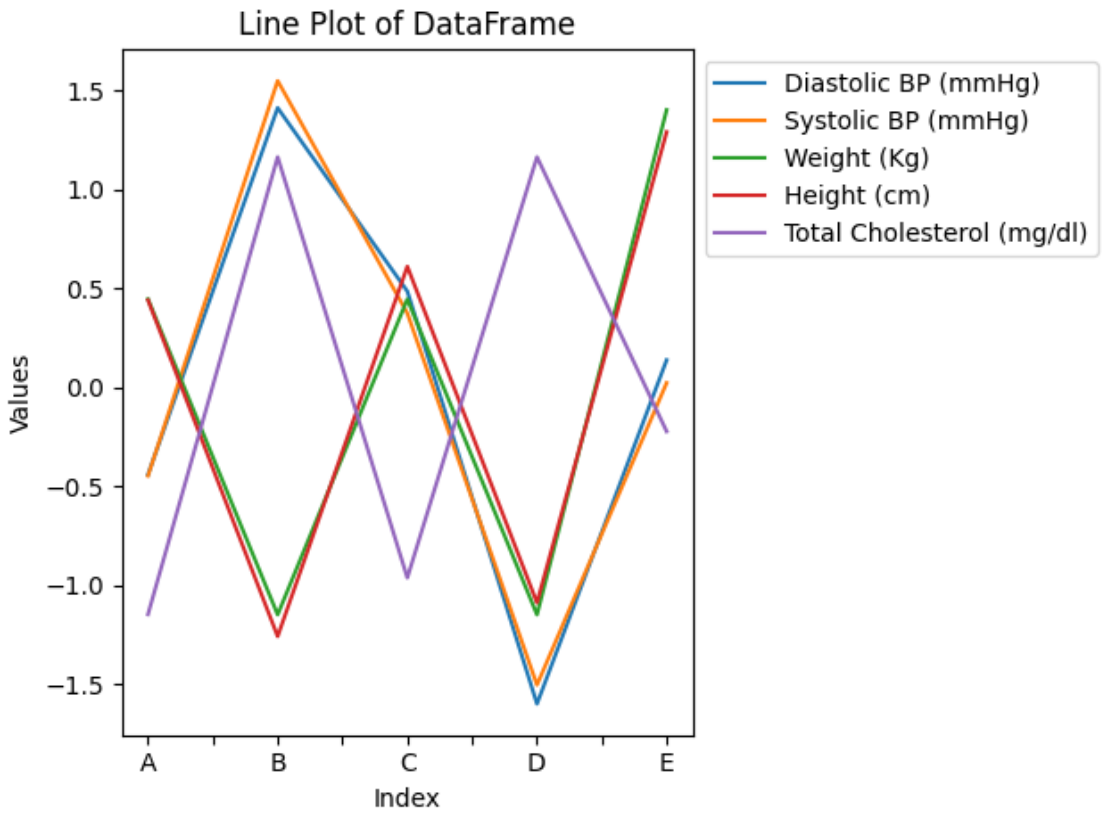
[17]:
```
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the data and transform it
df_normalized_z_score = pd.DataFrame(scaler.fit_transform(df.T).T, columns=df.
 ↪columns, index=df.index)
df_normalized_z_score
```

[17]:

	A	B	C	D	E
Diastolic BP (mmHg)	-0.440079	1.412885	0.486403	-1.598182	0.138972
Systolic BP (mmHg)	-0.446103	1.549620	0.375666	-1.502662	0.023479
Weight (Kg)	0.446304	-1.147638	0.446304	-1.147638	1.402669
Height (cm)	0.441758	-1.257311	0.611665	-1.087404	1.291293
Total Cholesterol (mg/dl)	-1.146184	1.164671	-0.961316	1.164671	-0.221842

When we standardize the data, ***variables that have a similar pattern will have a short distance to each other***.

[18]:
```
# Plotting the DataFrame using the default line plot
dft2 = df_normalized_z_score.T
dft2.plot()
plt.title('Line Plot of DataFrame')
plt.xlabel('Index')
plt.ylabel('Values')
# Place the legend outside the plot area to the right
# "loc" aligns the legend's upper left corner with the upper right corner of␣
 ↪the plot area
# "bbox_to_anchor" specifies the coordinates (relative to the plot area) where␣
 ↪the legend should be placed
plt.legend(loc='upper left', bbox_to_anchor=(1,1))
plt.tight_layout() # Adjust the layout to make room for the legend
plt.show()
```



[19]:
```
# Computing the distance matrix
distance_matrix = pd.DataFrame(
    # squareform converts a vector-form distance vector to a square-form␣
 ↪distance matrix,
    # and pdist computes the pairwise distances between observations in␣
 ↪n-dimensional space.
    squareform(pdist(df_normalized_z_score, metric='euclidean')),
    # Setting the columns and index of the resulting DataFrame to the index of␣
 ↪'df'
    # to maintain the same ordering and labeling.
    columns=df.index,
    index=df.index
)

# Displaying the computed distance matrix
distance_matrix
```

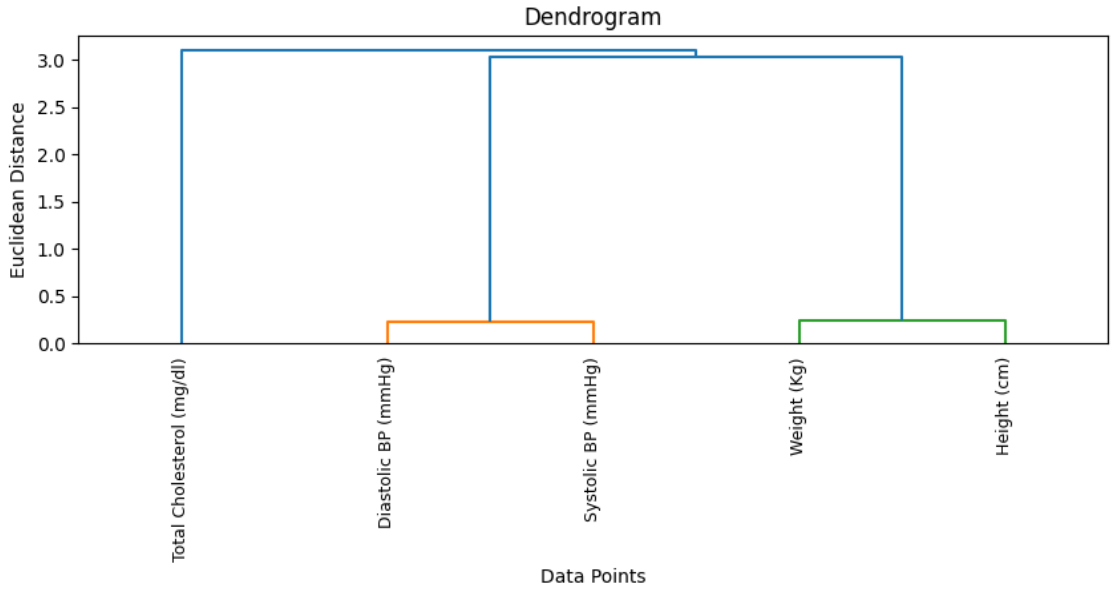
[19]:
```
                         Diastolic BP (mmHg)  Systolic BP (mmHg)
Diastolic BP (mmHg)                 0.000000            0.231210  \
Systolic BP (mmHg)                  0.231210            0.000000
```

```
Weight (Kg)                    3.023819          3.178803
Height (cm)                    3.084155          3.240779
Total Cholesterol (mg/dl)      3.227945          3.098495
```

	Weight (Kg)	Height (cm)	Total Cholesterol (mg/dl)
Diastolic BP (mmHg)	3.023819	3.084155	3.227945
Systolic BP (mmHg)	3.178803	3.240779	3.098495
Weight (Kg)	0.000000	0.235427	4.224926
Height (cm)	0.235427	0.000000	4.268870
Total Cholesterol (mg/dl)	4.224926	4.268870	0.000000

We can then generate the following dendrogram with the same method as we have seen previously. For example, the diastolic and systolic blood pressure variables now group together because they have a similar profile, which is also true for the variables weight and height.

[20]:
```
# Generating the linkage matrix
Z = linkage(df_normalized_z_score, 'single')
# Plotting the dendrogram
plt.figure(figsize=(10, 3))
dendrogram(Z, labels=labels, leaf_rotation=90, leaf_font_size=9)
plt.title("Dendrogram")
plt.xlabel("Data Points")
plt.ylabel("Euclidean Distance")
plt.show()
```



Linkage

The `linkage` function in the `scipy.cluster.hierarchy` module is used to perform hierarchical/agglomerative clustering. It takes a dataset and builds a hierarchical clustering using a bottom-

up approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.

Here's a description of the main components of the `linkage` function:

1. **Input Data**: The function can take an array of shape (n, m) as input, where n is the number of observations and m is the number of features. It can also take a condensed distance matrix, which is a one-dimensional array containing the upper triangular of the distance matrix (since it's symmetric).
2. **Method**: The `method` parameter specifies the linkage algorithm to use when merging clusters. Common methods include:
   - `'single'`: Nearest point algorithm, where the distance between two clusters is the minimum of all pairwise distances.
   - `'complete'`: Farthest point algorithm, where the distance between two clusters is the maximum of all pairwise distances.
   - `'average'`: Average of all pairwise distances.
   - `'centroid'`: Distance between the centroids of the clusters.
   - `'ward'`: Ward's method, which minimizes the variance within each cluster.
3. **Metric**: The `metric` parameter specifies the distance metric to use for computing the distance between observations. Common metrics include:
   - `'euclidean'`: Straight line distance between points.
   - `'cityblock'` or `'manhattan'`: Sum of absolute differences in their coordinates.
   - And many others, including `'minkowski'`, `'hamming'`, `'cosine'`, etc.
4. **Output**: The output of the `linkage` function is a linkage matrix. This matrix provides information about the hierarchical clustering structure. Each row represents a merge operation, with the following columns:
   - The first and second columns each contain the indices of the clusters that were merged.
   - The third column is the distance between the merged clusters.
   - The fourth column is the number of original observations in the newly formed cluster.

This linkage matrix can then be used to create a dendrogram, which is a tree-like diagram that shows the order of merges and the distance at which each merge occurred.

Here's a simple usage of the `linkage` function:

```
from scipy.cluster.hierarchy import linkage

# Assuming `data` is a 2D array or a condensed distance matrix
Z = linkage(data, method='complete', metric='euclidean')
```

`Z` is the linkage matrix that you can use to plot a dendrogram or cut the tree into clusters at a given distance.

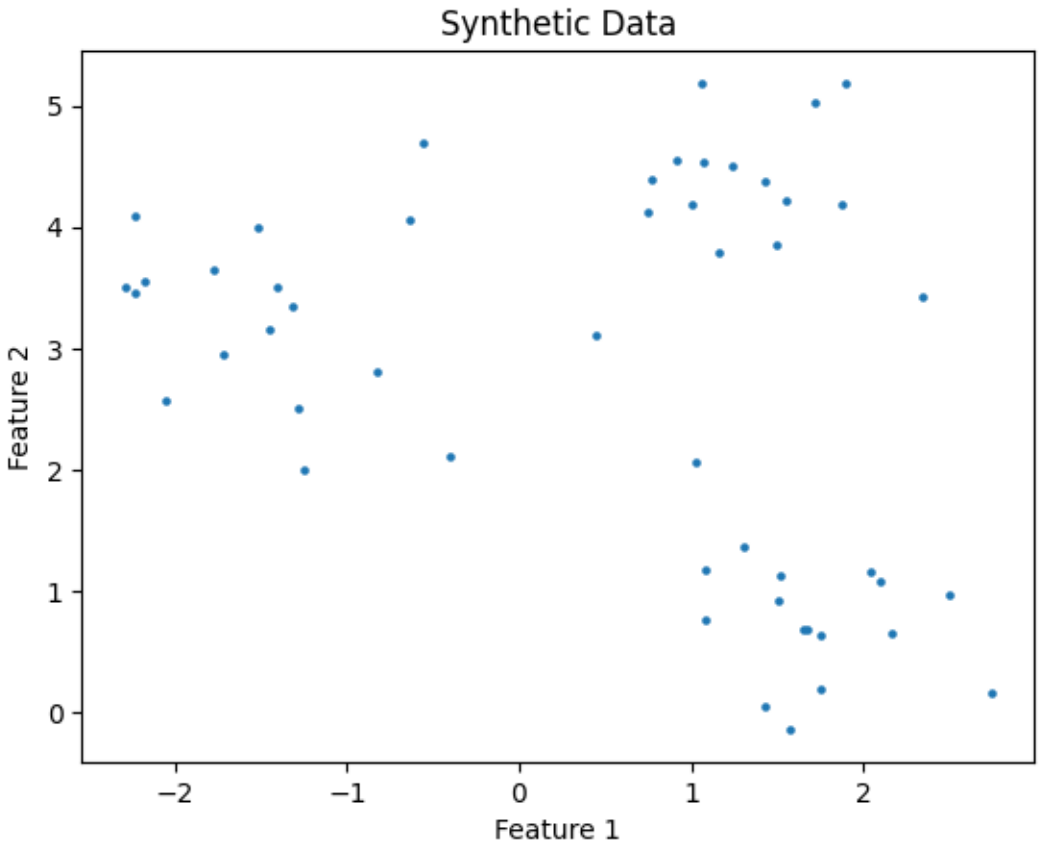
Exercise

Transpose the matrix and do the same exercise clustering with respect to the individuals instead of the variables.

1.4.1 A Simple Example with Synthetic Data

[21]:
```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generating synthetic data. Calls the make_blobs function to generate␣
 ↪synthetic data.
# - n_samples=50: Specifies the number of data points to generate.
# - n_features=2: Specifies the number of features (dimensions) for each data␣
 ↪point. In this case, it generates 2D data.
# - centers=3: Specifies the number of clusters to generate.
# - cluster_std=0.60: Specifies the standard deviation of the clusters. Higher␣
 ↪values indicate more spread-out clusters.
# - random_state=0: Sets the random seed for reproducibility. This ensures that␣
 ↪the generated data will be the same each time the code is run.
X, _ = make_blobs(n_samples=50, n_features=2, centers=3, cluster_std=0.60,␣
 ↪random_state=0)
```

[22]:
```
# Visualizing the data
plt.title("Synthetic Data")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.scatter(X[:, 0], X[:, 1], s=5)
plt.show()
```



Agglomerative Clustering with Scikit-Learn

The `AgglomerativeClustering` function in scikit-learn is an implementation of hierarchical agglomerative clustering, which is a type of clustering algorithm that builds a hierarchy of clusters. Here's a description of the function:

1. **Initialization**:
   - The `AgglomerativeClustering` function is initialized with parameters specifying the desired number of clusters or specifying linkage criteria (e.g., "ward", "complete", "average", or "single") and the distance metric to use (e.g., "euclidean", "manhattan", "cosine", etc.).
   - Parameters:
     - `n_clusters`: Specifies the number of clusters to form.
     - `linkage`: Specifies the linkage criterion to use. It determines the distance between newly formed clusters. Options include "ward", "complete", "average", and "single".
     - `affinity`: Specifies the distance metric to use. It is required if `linkage` is not "ward".
2. **Fitting the Model**:
   - The `fit` method is called on the initialized `AgglomerativeClustering` object to fit the model to the data.
   - Parameters:

- `X`: The input data to be clustered.

3. **Clustering**:
   - The algorithm iteratively merges the closest clusters based on the chosen linkage criterion and distance metric until the specified number of clusters is reached or until a stop condition is met.
   - At each step, it calculates the pairwise distances between clusters using the chosen distance metric and merges the closest clusters into larger clusters.
   - This process continues until the desired number of clusters is reached, resulting in a dendrogram or hierarchy of clusters.
4. **Predicting Cluster Assignments**:
   - After fitting the model, cluster assignments for the input data can be obtained using the `labels_` attribute.

fit_predict vs fit AND predict

In scikit-learn, both the `fit_predict` method and the combination of `fit` and `predict` methods are used for training models and making predictions. Here's an explanation of each method and their differences:

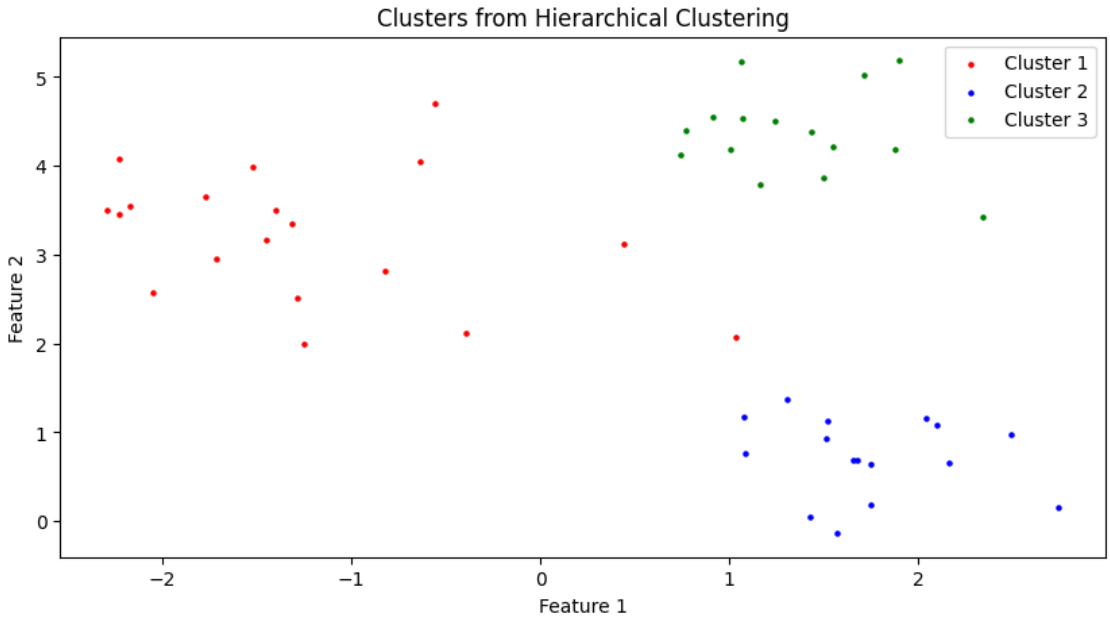
1. **fit_predict**:
   - The `fit_predict` method combines the training (fitting) and prediction steps into a single operation.
   - When calling `fit_predict`, the algorithm is first fitted to the training data, learning the underlying patterns and structure of the data.
   - After fitting, the algorithm immediately assigns each data point to a cluster, generating cluster labels as predictions.
   - This method is commonly used in clustering algorithms, where the primary goal is to both learn the clusters from the data and assign cluster labels to each data point simultaneously.
2. **fit and predict**:
   - The `fit` method is used to train the model on the training data. During this step, the algorithm learns the patterns and structure of the data.
   - The `predict` method is then used to make predictions on new, unseen data. This method takes the trained model and applies it to the new data, generating predictions based on the learned patterns.
   - In the context of clustering, `fit` is used to learn the clusters from the data, while `predict` is not typically used since clustering algorithms usually assign cluster labels during training and not on new, unseen data.

Key Differences: - **Usage**: `fit_predict` is used for clustering algorithms to both learn the clusters and assign cluster labels to the training data, while `fit` and `predict` are used separately for supervised learning tasks where predictions are made on new, unseen data. - **Output**: `fit_predict` directly outputs cluster labels for the training data, whereas `fit` and `predict` separately perform training and prediction steps, respectively. - **Convenience**: `fit_predict` can be more convenient when using clustering algorithms, as it combines both steps into a single call. However, `fit` and `predict` provide more flexibility, allowing for separate training and prediction phases.

In summary, `fit_predict` is specific to clustering algorithms and provides a convenient way to both train the model and generate cluster labels simultaneously. On the other hand, `fit` and `predict` are used separately for training and prediction in supervised learning tasks.

[23]:
```
from sklearn.cluster import AgglomerativeClustering

# Performing hierarchical clustering
hc    = AgglomerativeClustering(n_clusters=3, affinity='euclidean',␣
 ↪linkage='ward')
y_hc = hc.fit_predict(X)
```

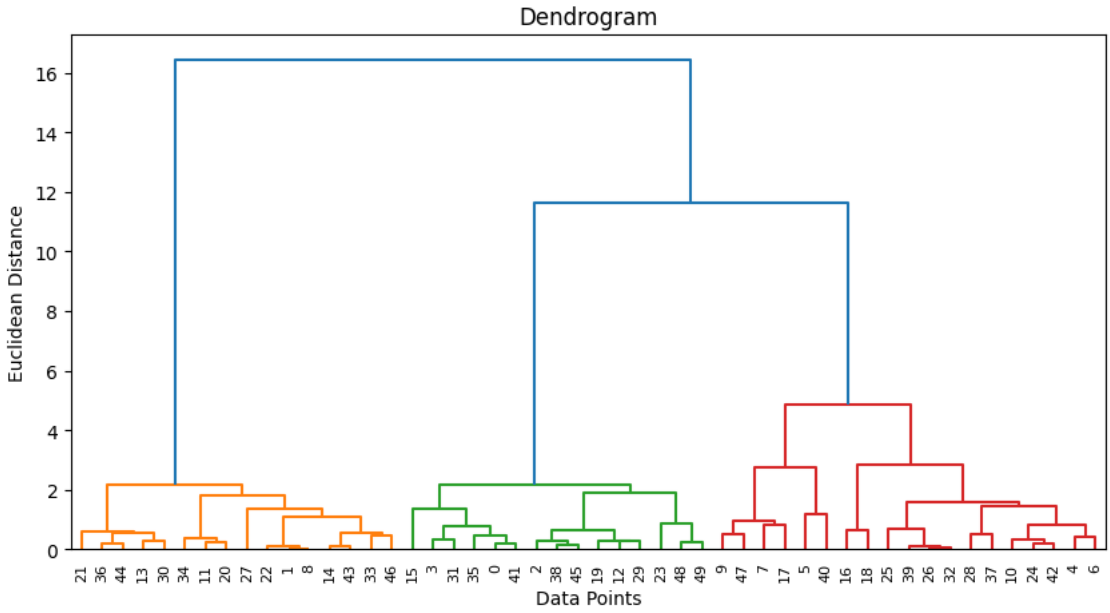
[24]:
```
# Visualizing the clusters
plt.figure(figsize=(10, 5))
plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s=5, c='red', label='Cluster 1')
plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s=5, c='blue', label='Cluster 2')
plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s=5, c='green', label='Cluster 3')
plt.title("Clusters from Hierarchical Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```



[25]:
```
from scipy.cluster.hierarchy import dendrogram, linkage

# Generating the linkage matrix
Z = linkage(X, 'ward')
```

[26]:
```
# Plotting the dendrogram
plt.figure(figsize=(10, 5))
dendrogram(Z)
```

```
plt.title("Dendrogram")
plt.xlabel("Data Points")
plt.ylabel("Euclidean Distance")
plt.show()
```



Advantages of Hierarchical Clustering:

- **No Need to Specify Number of Clusters**: Unlike K-means, hierarchical clustering does not require the user to specify the number of clusters beforehand.
- **Flexibility and Richness**: The hierarchical structure offers a more nuanced view of the data, as it captures relationships at different levels.
- **Easy to Interpret**: The dendrogram provides an intuitive and visual representation of the clustering process and data structure.

Challenges and Limitations:

- **Computational Complexity**: Especially for agglomerative methods, the computational complexity can be quite high for large datasets, making them less scalable than other clustering methods like K-means.
- **Sensitivity to Noise and Outliers**: Hierarchical clustering can be sensitive to noise and outliers, which can lead to misinterpretations of the data structure.
- **Irreversibility**: Once a step (merge or split) is done, it cannot be undone in later iterations, which might lead to suboptimal clustering in some cases.

Hierarchical clustering is widely used in various fields like bioinformatics for gene expression analysis, in marketing for customer segmentation, and in document clustering for information retrieval.

Its intuitive approach and the detailed insight it provides into the data structure make it a valuable tool in the unsupervised learning toolkit.

1.5 DBSCAN

Definition

DBSCAN, which stands for ***Density-Based Spatial Clustering of Applications with Noise***, is a popular clustering algorithm used in machine learning. It is a density-based clustering algorithm because it forms clusters based on dense regions of points. Here's an overview of how DBSCAN works:

1. **Core Points**: DBSCAN begins by identifying "core points," which are points that have at least a minimum number of other points (MinPts) within a certain radius (, epsilon). This radius can be thought of as a circle (or sphere in higher dimensions) drawn around each data point.
2. **Clusters**: Clusters are formed by connecting core points that are within distance of each other. If a core point is within this distance of another core point, they are part of the same cluster. This process continues until all core points are connected to clusters.
3. **Border Points**: There are also "border points," which are not core points (since they have fewer than MinPts within their neighborhood) but are within the radius of a core point and hence are included in the cluster.
4. **Noise**: Any point that is not a core point or a border point is considered "noise" and is not included in any cluster. This makes DBSCAN particularly good at handling outliers.

In this diagram, minPts = 4. Point A and the other red points are core points, because the area surrounding these points in an radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable.

Source:- Wikipedia

DBSCAN has several advantages:

- It **does not require specifying the number of clusters in advance**, unlike K-means.
- It can find **arbitrarily shaped** clusters.
- It has a notion of noise and is thus robust to outliers.

However, DBSCAN also has its drawbacks:

- It can struggle with datasets where clusters vary widely in density.
- Choosing the right and MinPts can be difficult without domain knowledge or some parameter tuning strategy.

In conclusion, DBSCAN is a powerful clustering algorithm when you have spatial data with clusters of similar density and you need an algorithm that can handle noise and outliers.

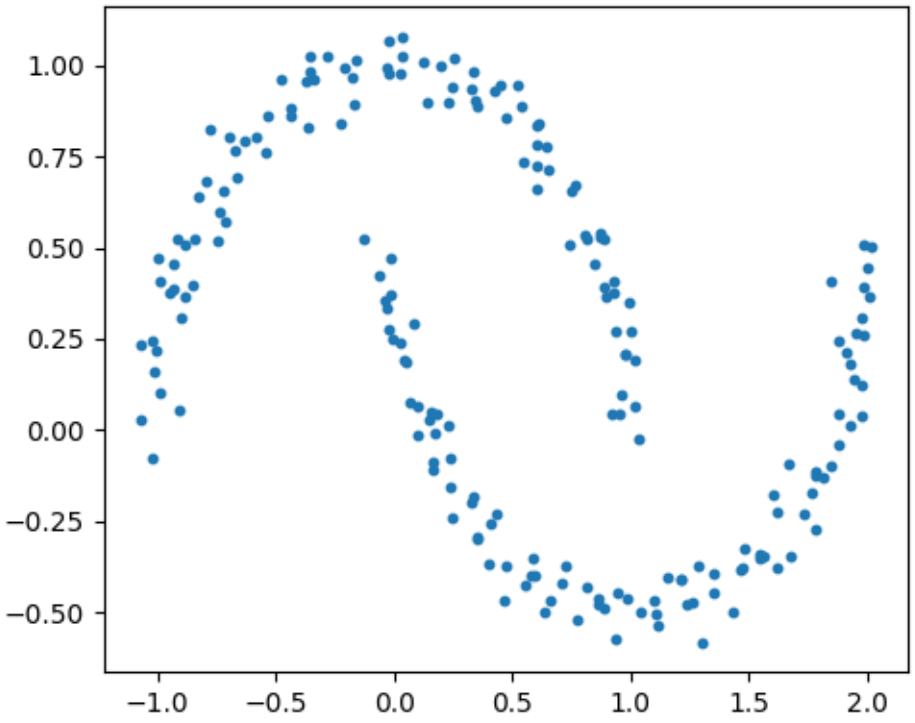
1.5.1 Use DBSCAN class in Scikit-Learn

The following code demonstrates the generation and visualization of synthetic moon-shaped data using scikit-learn's `make_moons` function and Matplotlib for plotting. Here's a breakdown of the code:

1. **Importing Libraries**:
   - `from sklearn.datasets import make_moons`: Imports the `make_moons` function from scikit-learn's `datasets` module. This function is used to generate synthetic data with moon-shaped clusters.
   - `import matplotlib.pyplot as plt`: Imports the Matplotlib library for plotting.
2. **Generating Synthetic Data**:
   - `X, y = make_moons(n_samples=200, noise=0.05, random_state=0)`: Calls the `make_moons` function to generate synthetic moon-shaped data.
     - `n_samples=200`: Specifies the number of data points to generate.
     - `noise=0.05`: Specifies the standard deviation of the Gaussian noise added to the data. Higher values result in noisier data.
     - `random_state=0`: Sets the random seed for reproducibility. This ensures that the generated data will be the same each time the code is run.
   - The generated data consists of two arrays:
     - `X`: An array of shape `(n_samples, 2)` containing the coordinates of the data points.
     - `y`: An array of shape `(n_samples,)` containing the labels of the data points (not used in this visualization).
3. **Visualizing the Data**:
   - `plt.scatter(X[:, 0], X[:, 1], s= 10)`: Creates a scatter plot of the synthetic data.
     - `X[:, 0]` and `X[:, 1]` represent the first and second features (columns) of the generated data, respectively.
     - `s=10`: Sets the size of the points in the scatter plot to 10 (small).
   - `plt.tight_layout()`: Adjusts the layout of the plot to prevent overlapping elements.
   - `plt.show()`: Displays the plot.

[27]:
```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=200,
                  noise=0.05,
                  random_state=0)
```

[28]:
```
plt.figure(figsize=(5, 4))
plt.scatter(X[:, 0], X[:, 1], s= 10)
plt.tight_layout()
plt.show()
```



Now we are ready to use the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm for clustering and visualizing the resulting clusters.

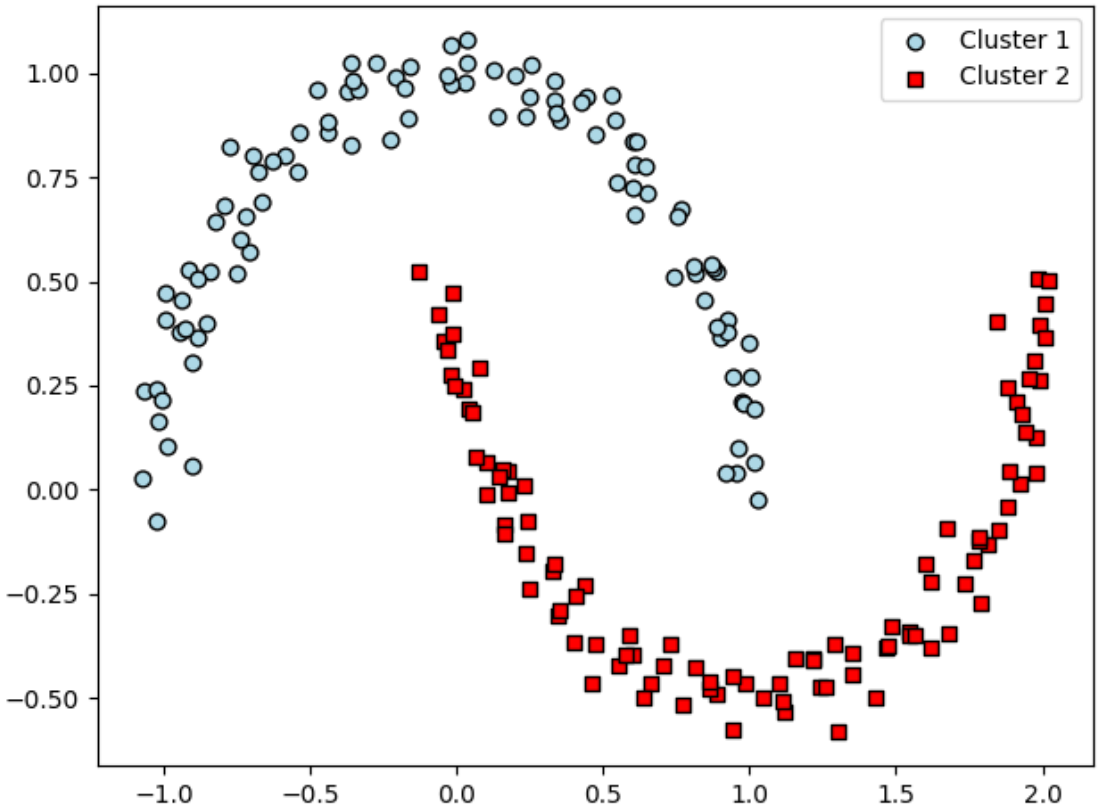
[33]:
```
from sklearn.cluster import DBSCAN

'''
creates an instance of the DBSCAN class with specific parameters:

- eps         = 0.2        : This sets the neighborhood radius (epsilon) to 0.2.
 ↪ Points within this distance are
                            considered neighbors.
- min_samples = 5          : This sets the minimum number of points required to␣
 ↪form a dense region (core points) to 5.
- metric      = 'euclidean': The distance metric for measuring the distance␣
 ↪between points is Euclidean distance.
'''
db = DBSCAN(eps         = .2,
            min_samples = 5,
            metric      = 'euclidean')
'''
The fit_predict method performs clustering on the dataset X and returns an␣
 ↪array y_db that contains
```

```
cluster labels for each point in X. Points labeled with the same number belong␣
 ↪to the same cluster,
while points labeled as -1 are considered noise
'''
y_db = db.fit_predict(X)
```

The remaining code uses matplotlib's `plt.scatter` to plot the points in the dataset `X`. It visualizes the clusters identified by DBSCAN:

- The first `plt.scatter` call plots the points belonging to the first cluster `(y_db == 0)`. These points are colored light blue, with circle markers (`'o'`), black edges, and size `40`.
- The second `plt.scatter` call plots the points belonging to the second cluster `(y_db == 1)`. These are shown in red, with square markers (`'s'`), black edges, and size `40`.
- `plt.legend()` adds a legend to the plot.
- `plt.tight_layout()` adjusts the plot layout for better readability.
- `plt.show()` displays the plot.

[34]:
```
plt.scatter(X[y_db==0, 0],X[y_db==0,␣
 ↪1],c='lightblue',edgecolor='black',marker='o',s=40,label='Cluster 1')
plt.scatter(X[y_db==1, 0],X[y_db==1, 1],c='red',edgecolor='black',marker='s',s=␣
 ↪40,label='Cluster 2')
plt.legend()
plt.tight_layout()
plt.show()
```



1.6 References and Credits

[1] - Géron A. - "Hands On Machine Learning with Scikit-Learn Keras and Tensorflow", 2nd Ed, O'Reilly

[2] - Irani J. et al. - "Clustering Techniques and the Similarity Measures used in Clustering: A Survey", *International Journal of Computer Applications (0975 – 8887), Volume 134 – No.7, January 2016.*

[3] - Hull J. C. - "Machine Learning in Business: An Introduction to the World of Data Science", Amazon, 2019.

Other Usefull link:

Equivalent of Elbow Method

1.7 Exercises

1.7.1 Exercise 1: Implementing K-Means Clustering with Synthetic Data

In this exercise, you will get hands-on experience with unsupervised machine learning, focusing on the K-Means clustering method. You will generate synthetic data, apply the K-Means algorithm to identify clusters, and make predictions for new data points. Follow the steps below:

Part 1: Data Generation

1. **Generate Synthetic Data**: Use the `make_blobs` function from `sklearn.datasets` to create a dataset suitable for demonstrating a K-Means clustering. Your dataset should have the following characteristics:

   - 2000 samples.
   - 5 pre-defined cluster centers.
   - Standard deviations for each cluster to control the spread of the points.
   - A random state for reproducibility.

   Hint: Define the cluster centers and standard deviations as numpy arrays before generating the data.

2. **Visualize the Generated Data**: Plot the generated dataset using a scatter plot. Ensure each point's color corresponds to its actual cluster to visually assess the distribution of the data.

Part 2: Applying K-Means Clustering

1. **Initialize the K-Means Model**: Initialize a K-Means clustering model from `sklearn.cluster` with the following parameters:

   - Number of clusters `k` equal to 5, matching the number of pre-defined cluster centers.
   - A random state for reproducibility.

2. **Fit the Model and Predict**: Fit the K-Means model to the generated dataset and then use the fitted model to predict the cluster labels for the dataset.

3. **Access Cluster Information**:

   - Retrieve the coordinates of the cluster centers from the fitted model.
   - Verify that the predicted labels match the labels assigned by the model during fitting.

Part 3: Making Predictions

1. **Predict New Data Points**: With the fitted K-Means model, predict the cluster labels for new data points. Consider the following new points for prediction: `[[0, 2], [3, 2], [-3, 3], [-3, 2.5]]`.

Deliverables:

- A scatter plot of the generated dataset with points colored according to their actual clusters.
- The coordinates of the cluster centers identified by the K-Means model.
- Predicted cluster labels for the new data points.

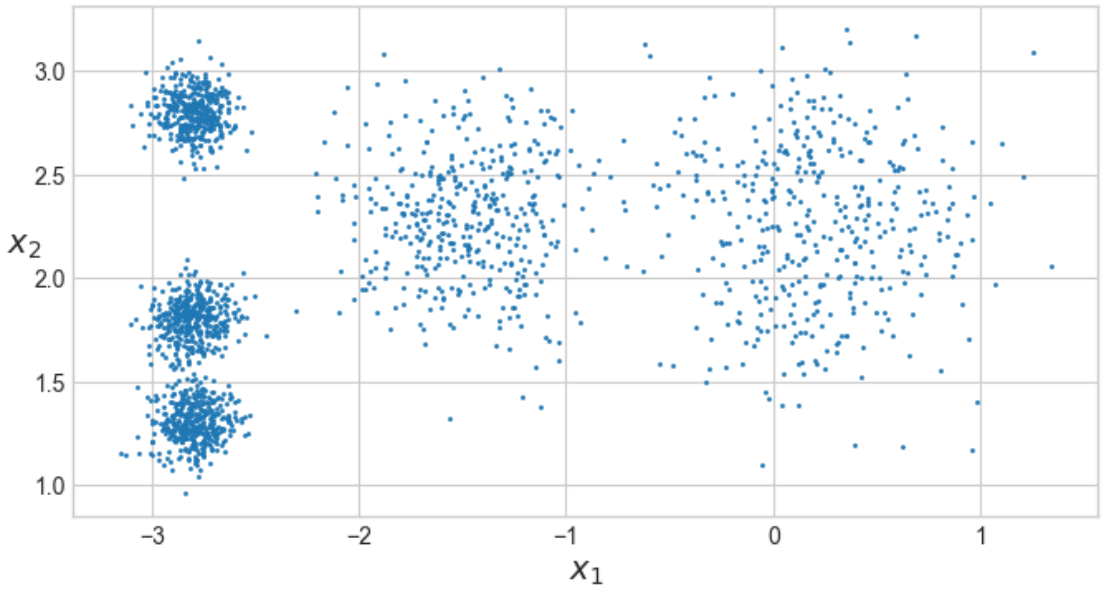
Learning Objective:

By completing this exercise, you will understand how to generate synthetic data for clustering, apply the K-Means algorithm to identify clusters within data, and use the model to make predictions for new data points. This exercise will reinforce concepts of unsupervised machine learning and the practical use of the K-Means clustering method.

SOLUTION

[218]:
```
# Import necessary libraries
from sklearn.datasets import make_blobs
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Define the centers of the blobs to be generated
blob_centers = np.array(
    [[ 0.2,  2.3],  # Center of the first blob
     [-1.5 ,  2.3], # Center of the second blob
     [-2.8,  1.8],  # Center of the third blob
     [-2.8,  2.8],  # Center of the fourth blob
     [-2.8,  1.3]]) # Center of the fifth blob

# Define the standard deviation of the blobs
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])  # Spread of each blob

# Generate synthetic data with predefined blob centers and standard deviations
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)

# Define a function to plot the clusters
def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)  # Plot points with colors based on␣
 ↪their cluster
    plt.xlabel("$x_1$", fontsize=14)  # x-axis label
    plt.ylabel("$x_2$", fontsize=14, rotation=0)  # y-axis label

# Plot the generated clusters
plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.show()

# Define the number of clusters to identify
k = 5

# Initialize the KMeans clustering model with 5 clusters and a fixed random␣
 ↪state for reproducibility
kmeans = KMeans(n_clusters=k, random_state=42)

# Fit the model to the data and simultaneously predict the cluster labels for␣
 ↪the input data
y_pred = kmeans.fit_predict(X)

# Access the coordinates of the cluster centers determined by the KMeans␣
 ↪algorithm
kmeans.cluster_centers_
```

```
# Access the labels (clusters) assigned to each data point
kmeans.labels_

# Define new data points for which we want to predict the cluster membership
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])

print("Predict the cluster labels for the new data points\n")
kmeans.predict(X_new)
```




```
Predict the cluster labels for the new data points
```

[218]:
```
array([1, 1, 2, 2])
```

1.7.2 Exercise 2: Implementing Hierarchical Clustering with Kaggle Dataset

Downloaded the following dataset from Kaggle. This dataset contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. It also gives the percent of the population living in urban areas.

SOLUTION

[219]:
```
from sklearn.preprocessing import StandardScaler
scaler= StandardScaler()

import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering
```

[220]:
```
crime = pd.read_csv("./data/USArrests.csv")
crime.head()
```

[220]:

	rownames	Murder	Assault	UrbanPop	Rape
0	Alabama	13.2	236	58	21.2
1	Alaska	10.0	263	48	44.5
2	Arizona	8.1	294	80	31.0
3	Arkansas	8.8	190	50	19.5
4	California	9.0	276	91	40.6

[222]:
```
# Renaming the first column.
crime = crime.rename(columns={'rownames':'State'})
crime.head()
```

[222]:

	State	Murder	Assault	UrbanPop	Rape
0	Alabama	13.2	236	58	21.2
1	Alaska	10.0	263	48	44.5
2	Arizona	8.1	294	80	31.0
3	Arkansas	8.8	190	50	19.5
4	California	9.0	276	91	40.6

[223]:
```
# Let's get some statistics summary
crime.describe()
```

[223]:

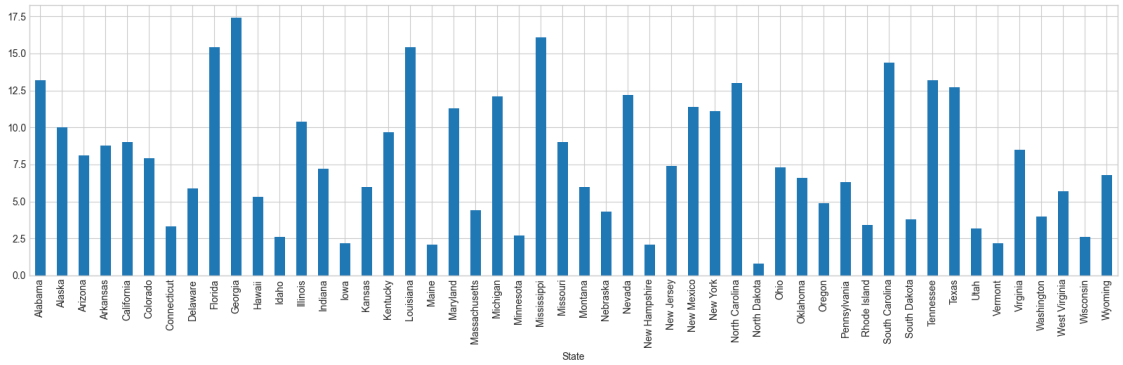
	Murder	Assault	UrbanPop	Rape
count	50.00000	50.000000	50.000000	50.000000
mean	7.78800	170.760000	65.540000	21.232000
std	4.35551	83.337661	14.474763	9.366385
min	0.80000	45.000000	32.000000	7.300000
25%	4.07500	109.000000	54.500000	15.075000
50%	7.25000	159.000000	66.000000	20.100000
75%	11.25000	249.000000	77.750000	26.175000
max	17.40000	337.000000	91.000000	46.000000

[224]:
```
# Let's check for missing values
crime.isnull().sum()
```

[224]:
```
State       0
Murder      0
Assault     0
UrbanPop    0
Rape        0
dtype: int64
```

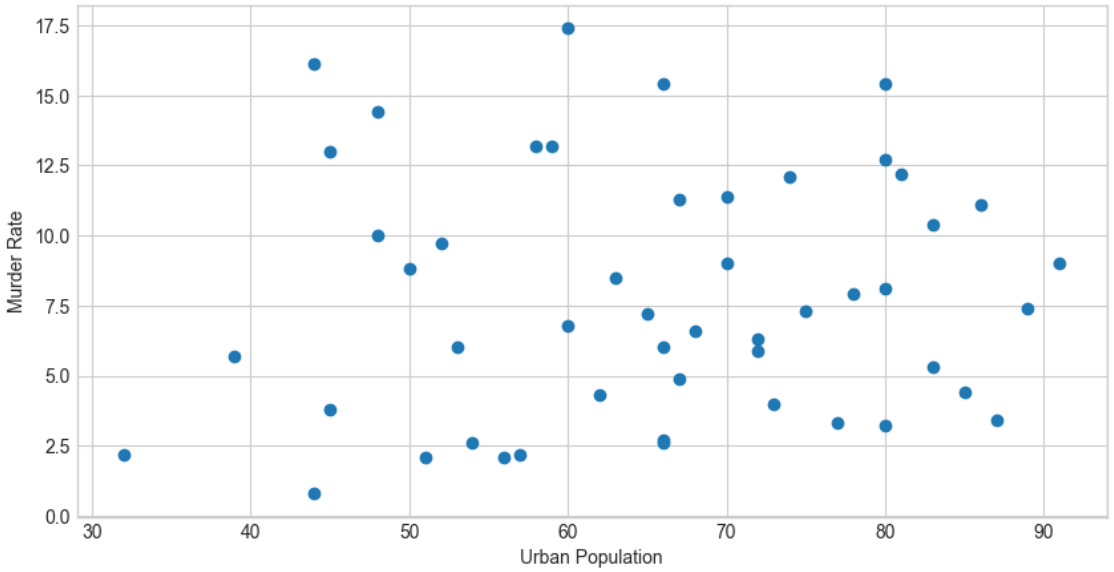
[225]:
```
plt.figure(figsize=(20,5))
# Set 'State' as index
crime.set_index('State', inplace=True)
crime['Murder'].plot(kind='bar')
```

[225]: <Axes: xlabel='State'>



[226]:
```
plt.figure(figsize=(10,5))
plt.scatter('UrbanPop','Murder',data=crime)
plt.xlabel('Urban Population')
plt.ylabel('Murder Rate')
```

[226]: Text(0, 0.5, 'Murder Rate')

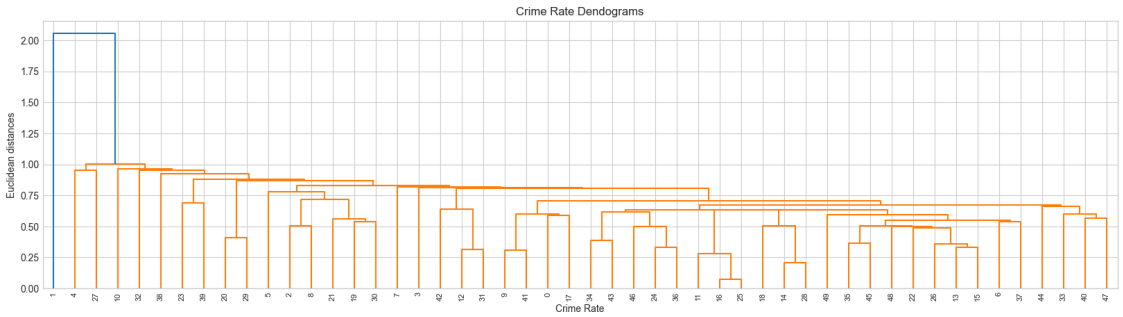


[227]:
```
data = crime.iloc[:,1:].values
scaled_data = scaler.fit_transform(data)
```

[228]:
```
# SINGLE LINKAGE
```

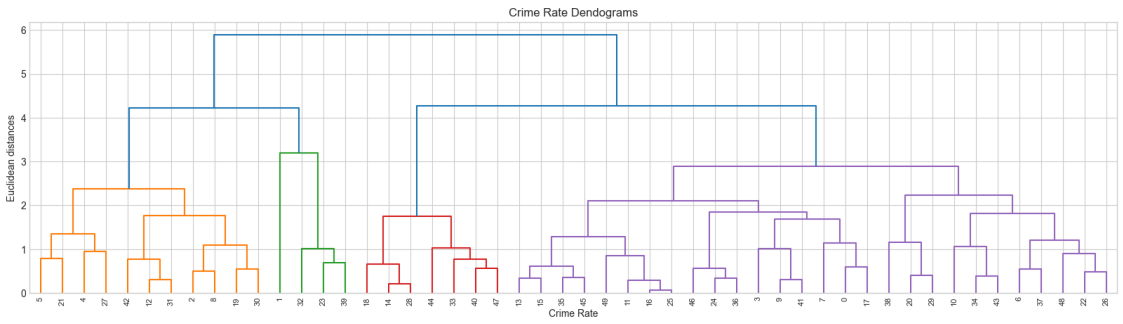
```
plt.figure(figsize=(20,5))
plt.title("Crime Rate Dendograms")
dend = sch.dendrogram(sch.linkage(scaled_data, method='single'))
plt.xlabel('Crime Rate')
plt.ylabel('Euclidean distances')
```

[228]: Text(0, 0.5, 'Euclidean distances')



[229]:
```
# COMPLETE LINKAGE

plt.figure(figsize=(20,5))
plt.title("Crime Rate Dendograms")
dend = sch.dendrogram(sch.linkage(scaled_data, method='complete'))
plt.xlabel('Crime Rate')
plt.ylabel('Euclidean distances')
```

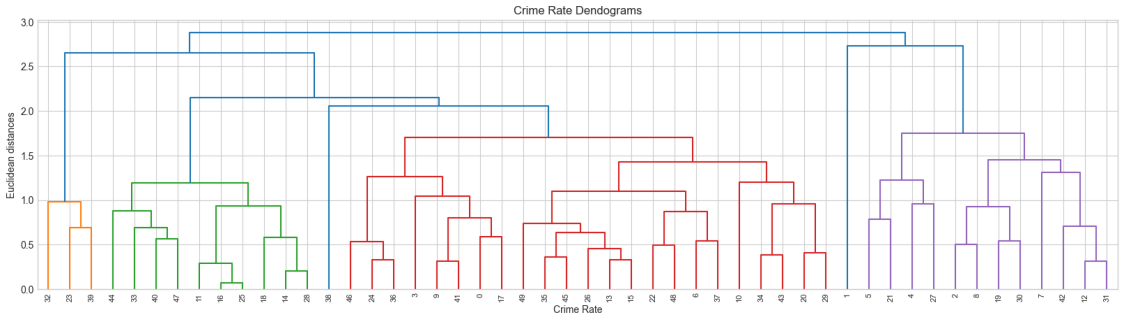
[229]: Text(0, 0.5, 'Euclidean distances')



[230]:
```
# AVERAGE LINKAGE

plt.figure(figsize=(20,5))
plt.title("Crime Rate Dendograms")
dend = sch.dendrogram(sch.linkage(scaled_data, method='average'))
```

```
plt.xlabel('Crime Rate')
plt.ylabel('Euclidean distances')
```

[230]: `Text(0, 0.5, 'Euclidean distances')`



The single linkage type will produce dendrograms which are not structured properly, whereas complete or average linkage will produce clusters which have a proper tree-like structure.

1.8 Appendix

This Python code defines a set of functions for plotting data points, centroids, and decision boundaries associated with a clustering model, specifically using K-Means clustering. It visualizes the clustering results, including the data points, the centroids of the clusters, and the decision boundaries that separate the clusters. Here's a detailed explanation:

`plot_data(X)`

- **Purpose**: Plots the data points.
- **Parameters**: `X` - a 2D numpy array where each row represents a data point, and each column represents a feature (dimension).
- **Functionality**: It plots the data points in a 2-dimensional space, using black dots (`'k.'`) with a marker size of 2.

`plot_centroids(centroids, weights=None, circle_color='w', cross_color='k')`

- **Purpose**: Plots the centroids of the clusters.
- **Parameters**:
  - `centroids`: a 2D numpy array where each row is the coordinates of a centroid.
  - `weights`: (optional) an array of weights for the centroids. If provided, only the centroids with weights greater than one-tenth of the maximum weight are plotted.
  - `circle_color`: the color of the circles drawn around the centroids.
  - `cross_color`: the color of the crosses marking the centroids.
- **Functionality**: This function first checks if weights are provided and filters the centroids accordingly. It then plots the centroids with circle and cross markers to highlight them. Circles are drawn with specified `circle_color`, and crosses with `cross_color`. Circles have a z-order of 10 (which determines drawing order), making them appear on top of other elements except for crosses, which have a z-order of 11.

`plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=True, show_xlabels=True, show_ylabels=True)`

- **Purpose**: Plots the decision boundaries for a clustering model, along with the data points and optionally the centroids.
- **Parameters**:
  - `clusterer`: The clustering model that provides the `.predict` method to determine the cluster for each point in the space.
  - `X`: 2D numpy array of data points.
  - `resolution`: The resolution of the grid used to plot the decision boundaries.
  - `show_centroids`: Whether to plot the centroids.
  - `show_xlabels`, `show_ylabels`: Whether to show labels on the x and y axes.
- **Functionality**: The function creates a meshgrid spanning the data's extent, with an additional margin. It then uses the `predict` method of the `clusterer` to classify each point in the meshgrid, thereby determining the decision boundaries. These boundaries are visualized using contour plots (`contourf` for filled contours indicating different clusters and `contour` for boundary lines). Data points and centroids (if `show_centroids` is True) are plotted on top of this. Labels on the axes are managed based on `show_xlabels` and `show_ylabels` flags.

Finally, a figure is created with a specified size, and `plot_decision_boundaries` function is called with the KMeans model `kmeans` and the data `X` to plot everything together. This visualization helps in understanding how the KMeans algorithm has partitioned the space and where the centroids of these clusters are located relative to the data points.

[231]:
```
# Necessary imports for handling data and plotting
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
#_____________________________________________________________________________________________
#
# Function to plot data points
def plot_data(X):
    """
    Plots the data points.

    Parameters:
    - X: 2D numpy array of data points, where each row represents a data point
         and the columns represent the features of the data point.
    """
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)  # Plot data points as black␣
 ↪dots


#_____________________________________________________________________________________________
#
# Function to plot centroids of the clusters
def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k'):
    """
    Plots the centroids of clusters with optional weighting.
```

```
    Parameters:
    - centroids: 2D numpy array where each row represents the coordinates of a␣
 ↪centroid.
    - weights: Optional numpy array of weights for the centroids. Centroids␣
 ↪with weights
                greater than one-tenth of the maximum weight are plotted.
    - circle_color: Color of the circles around centroids.
    - cross_color: Color of the crosses marking the centroids.
    """
    if weights is not None:  # Filter centroids based on weights, if weights␣
 ↪are provided
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=35, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)  # Plot circles for␣
 ↪centroids
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=2, linewidths=12,
                color=cross_color, zorder=11, alpha=1)  # Plot crosses for␣
 ↪centroids

#_____________________________________________________________________________________________
#
# Function to plot decision boundaries of clusters
def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=True,
                             show_xlabels=True, show_ylabels=True):
    """
    Plots the decision boundaries for a clustering model, the data points, and␣
 ↪optionally the centroids.

    Parameters:
    - clusterer: Clustering model with a .predict method to classify points.
    - X: 2D numpy array of data points.
    - resolution: The resolution of the grid for plotting decision boundaries.
    - show_centroids: Boolean indicating whether to plot centroids.
    - show_xlabels, show_ylabels: Booleans indicating whether to show labels on␣
 ↪the x and y axes.
    """
    # Determine the plotting area based on the data points
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    # Create a meshgrid for the plotting area
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    # Predict cluster for each point in the meshgrid
```

```
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot filled contours and decision boundaries
    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]), cmap="Pastel2")
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]), linewidths=1,␣
↪colors='k')
    plot_data(X)  # Plot the data points on top
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)  # Plot centroids if␣
↪specified

    # Manage axis labels based on flags
    if show_xlabels:
        plt.xlabel("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)

# Create a figure for plotting
plt.figure(figsize=(8, 4))
# Plot decision boundaries using the previously defined function
plot_decision_boundaries(kmeans, X)  # Assumes 'kmeans' model and dataset 'X'␣
↪are defined earlier
plt.show()  # Display the plot
```